Akademia Górniczo-Hutnicza
im. Stanisława Staszica
w Krakowie

Praca dyplomowa

magisterska

Interfejs graficzny do budowy aplikacji komponentowych zapewniający weryfikację semantyczną

Maciej Kwiecień, Jan Rachwalik

Kierunek: Informatyka
Specjalność: Systemy Rozproszone
i Sieci Komputerowe

Nr albumu: 116969 (MK)
117001 (JR)

Promotor
prof. dr hab. inż. Krzysztof Zieliński

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

Kraków, 2007

Oświadczenie autorów

Oświadczamy,świadomi odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonaliśmy osobiście i samodzielnie (w zakresie wyszczególnionym we wstępie) i że nie korzystaliśmy ze źródeł innych niż podane w pracy.

.....
(Podpis autora)

.....
(Podpis autora)

Oświadczenie promotora

Oświadczam, że praca spełnia wymogi stawiane pracom magisterskim.

.....
(Podpis promotora)

Pracę tą pragniemy zadedykować naszym rodzicom, jednocześnie dziękując za otrzymane z ich strony wsparcie w ciągu całego procesu edukacji.

Dziękujemy również prof. dr hab. inż. Krzysztofowi Zielińskiemu i dr inż. Łukaszowi Czekierdzie, bez których pomocy praca ta nie mogłaby powstać.

Spis treści

Rozdział 1

Wstęp

Tworzenie aplikacji rozproszonych wiąże się z dużą złożonością, co powoduje, że budowanie takich systemów jest pracochłonne i długotrwałe. Interesującym aspektem, w rozwoju środowisk rozproszonych jest fakt, że programiści muszą rozwiązywać te same problemy niższych warstw dla każdego systemu. Mamy tu na myśli:

- bezpieczeństwo,
- obsługa zdarzeń,
- transakcyjność,
- persystencję.

Dobre rozwiązanie przedstawionych powyżej kwestii gwarantuje niezawodność tworzonej aplikacji. Powstaje więc pytanie, czy te typowe aspekty systemów rozproszonych mogłyby być łatwe w implementacji i rozszerzalne bezproblemowo. Odpowiedzią na takie potrzeby jest specyfikacja *CORBA Component Model*. Model *CCM* jest częścią specyfikacji środowiska *CORBA* 3.0. Model ten opisuje tworzenie i osadzanie rozproszonych aplikacji opartych o obiekty środowiska *CORBA*. Model *CCM* jest podobny do *Enterprise Java Beans* poprzez użycie wzorców projektowych oraz ułatwień umożliwiających automatyczne generowanie kodu. Omawiany model udostępnia kontenery zajmujące się podstawowymi serwisami wykorzystywanymi w systemach rozproszonych.

Proces stworzenia aplikacji rozproszonej w modelu *CCM* jest na tyle złożony, że istnieje potrzeba użycia programów wspomagających ten proces. Jednym z kluczowych etapów w tworzeniu systemu rozproszonego jest etap asemblacji, w którym następuje konfiguracja komponentów, łączenie ich portów oraz przypisanie poszczególnym komponentom serwerów, w których mają zostać osadzone. Etap ten kończy się powstaniem pliku archiwum zawierającym m.in. deskryptor asemblacji (plik o rozszerzeniu *.CAD*) zawierający przeprowadzoną konfigurację.

Ręczne tworzenie deskryptorów o postaci pliku *XML* jest możliwe, ale jest mozolne i sprzyja powstawaniu dużej ilości błędów. Widać więc potrzebę stworzenia aplikacji pomocnej przy tworzeniu wspomnianych plików. Na rynku istnieje już narzędzie o nazwie Cadena posiadające funkcję generowania deskryptora asemblacji. Jest to środowisko do tworzenia systemów *CCM*, które wspomaga:

- definiowanie zależności między komponentami, ich stanów oraz przejść między nimi,
- analizę zależności między komponentami,
- kontrolę infrastruktury modeli opartych na zdarzeniowej komunikacji między komponentami,
- generacje kodu stub-ów i skeleton-ów przy użyciu OpenCCM (w wersji 0.7 - aktualna wersja OpenCCM 0.9),
- budowanie systemu (*assembly*),
- rozmieszczanie komponentów w kontenerach (*deployment*).

Narzędzie zostało zaprojektowane jako rozszerzenie do środowiska *Eclipse*. Aplikacja ta posiada jednak wady, które uniemożliwiają wykorzystanie jej przy tworzeniu systemów:

- trudne, nieprzyjazne dla użytkownika środowisko - ogromne problemy z assemblacją,
- wprowadza dużo własnych formatów plików,
- oparta jest na Javie 1.4.

Tak więc celem naszej pracy jest stworzenie przyjaznego dla użytkownika narzędzia do tworzenia deskryptora assemblacji (.cad) w trybie graficznym na zasadzie łączenia portów komponentów.

Niniejsza praca składa się z sześciu rozdziałów.

Rozdział 1. stanowi wprowadzenie do pracy.

Zadaniem rozdziału 2. jest przedstawienie czytelnikowi modelu *CORBA Component Model*. Ze względu na złożoność modelu przedstawiamy tylko najważniejsze jego aspekty; szczególny nacisk kładziemy na elementy specyfikacji wykorzystywane przez nasz edytor.

Rozdział 3. opisuje bibliotekę *GEF,* która umożliwia tworzenie edytorów graficznych w środowisku *Eclipse* i znacznie ułatwiła stworzenie aplikacji do graficznego tworzenia deskryptora asemblacji. Przedstawiamy proces tworzenia edytora oraz technikę programowania przy wykorzystaniu biblioteki *GEF.*

W rozdziale 4. przedstawiamy opis skonstruowanego graficznego narzędzia generującego deskryptor asemblacji. Aplikację tę nazwaliśmy *CGE (CAD Graphical Editor).* Opisujemy postawione narzędziu wymagania wraz ze sposobem ich realizacji.

Rozdział 5. służy za podręcznik użytkownika i przedstawia korzystanie z aplikacji krok po kroku. Oprócz opisu samego narzędzia pojawia się proces konfiguracji środowiska *OpenCCM.*

Rozdział 6. stanowi podsumowanie pracy oraz ocenę stworzonej aplikacji.

Rozdział 2

CCM - CORBA Component Model

W niniejszym rozdziale zostanie przedstawiony model tworzenia rozproszonych aplikacji komponentowych zdefiniowany przez grupę *The Object Management Group* (*OMG).* Opis zawiera jedynie najistotniejsze elementy specyfikacji wymagane do zrozumienia dalszej częsci pracy. W drugiej częsci rozdziału znajduje się krótki wstęp do używanej implementacji modelu, środowiska *OpenCCM.* Na koniec opisano deskryptor asemblacji, którego generowanie i walidacja jest podstawą zaprojektowanej i zaimplementowanej aplikacji dołączonej do pracy.

2.1. Wprowadzenie do CORBA Component Model

Obecnie coraz większy nacisk w programowaniu kładziony jest na szybkość i prostotę tworzenia niezawodnych aplikacji. Konkurencja na rynku tworzenia oprogramowania wymaga, aby tworzone aplikacje charakteryzowały się wysoką wydajnością, powstawały przy wykorzystywaniu istniejącego kodu oraz mogły być rozwijane w łatwy sposób. W celu zmniejszenia kosztów oraz czasu projektowania i budowania programów udostępnia się możliwość projektowania, tworzenia, grupowania oraz wdrażania aplikacji korporacyjnych w oparciu o komponenty. Jednym z modeli rozwijających programowanie komponentowe jest *CORBA Component Model (CCM).*

2.1.1. Cele, zalety i pola eksploatacji środowiska CCM

W tworzeniu oprogramowania znaczenia nabiera programowanie wykorzystujące asemblację w miejsce standardowej inżynierii oprogramowania. Dąży się do tego, aby standardowe problemy programistyczne były rozwiązywane na zasadzie produkowania, tzn. łączenia istniejących elementów ze sobą oraz dostosowywania ich do konkretnego przeznaczenia. Omawiana technika asemblacji prowadzi do obniżenia umiejętności

wymaganych od programisty, pozwala skupić się na dziedzinie problemu, a nie na sposobie jego rozwiązywania oraz pozwala na szybkie tworzenie wydajnych aplikacji.

Rozbudowane aplikacje korporacyjne korzystają z wielowarstwowych modeli. Role serwera i klienta są ściśle odseparowane od siebie. Po stronie klienta może działać klient WWW, aplet lub aplikacja kliencka. Zadaniem klienta jest tylko prezentacja danych generowanych przez stronę serwerową. Cała logika aplikacji mieści się po stronie serwera i jest dzielona na komponenty zgodnie z ich przeznaczeniem i funkcjami, a poszczególne komponenty są umieszczane na różnych komputerach zależnie od tego, do jakiej warstwy aplikacji należą.

Rozproszony, zorientowany obiektowo model *CORBA Component Model* zawdzięcza technoligii CORBA:

- różnorodność - dzięki wykorzystaniu języka *OMG Interface Definition Language (OMG IDL)* do opisu interfejsów zdalnych obiektów CORBA, interfejsy te są uniezależnione od systemu operacyjnego,
- przenośność - możliwość współpracy różnych języków programowania,
- interoperacyjność (*ang. interoperability*) - dzięki użyciu protokołu komunikacji *IIOP (Internet Inter-ORB Protocol)* implementującego standard *GIOP (General Inter-ORB Protocol)* możliwe jest stosowanie rozwiązań dostarczanych przez różnych producentów,
- możliwość wykorzystania różnych modeli wywołania: *Static/Stub Invocation Interface (SII), Dynamic Invocation Interface (DII), Asynchronous Method Invocations (AMI),*
- *ORB (Object Request Broker)* pośredniczy w komunikacji pomiędzy rozproszonymi obiektami,
- jawne określenie właściwości niefunkcjonalnych tj.: czas życia, aktywacja/deaktywacja, notyfikacje, transakcje, bezpieczeństwo, itp.

Model *CCM* dedykowany jest dla:

- rozproszonych aplikacji korporacyjnych,
- aplikacji naukowych wykorzystujących zdalne zasoby,
- oprogramowania z dziedziny medycyny - zdalne wykorzystanie aparatury.

2.1.2. Etapy tworzenia aplikacji CCM

Tworzenie aplikacji CCM jest wieloetapowe. Poszczególne etapy pozostają w pewnych zależnościach pomiędzy sobą. Proces rozwijania aplikacji przedstawia się następująco:

- **Projektowanie komponentów** - definicja komponentów i ich obiektów *home* za pomocą rozszerzeń języka OMG IDL 3.0. W tym etapie powstają:
  — deskryptory komponentów,
  — klasy typu *stub* oraz *skeleton.*
  Etap wykonywany przez **projektanta.**
- **Implementacja funkcjonalności komponentów** - tworzenie operacji logiki biznesowej aplikacji. Tworzone operacje implementują interfejsy zdefiniowane przez projektanta. W tej fazie powstają:
  — skompilowane klasy komponentów,

— wzbogacone deskryptory XML opisujące komponenty.

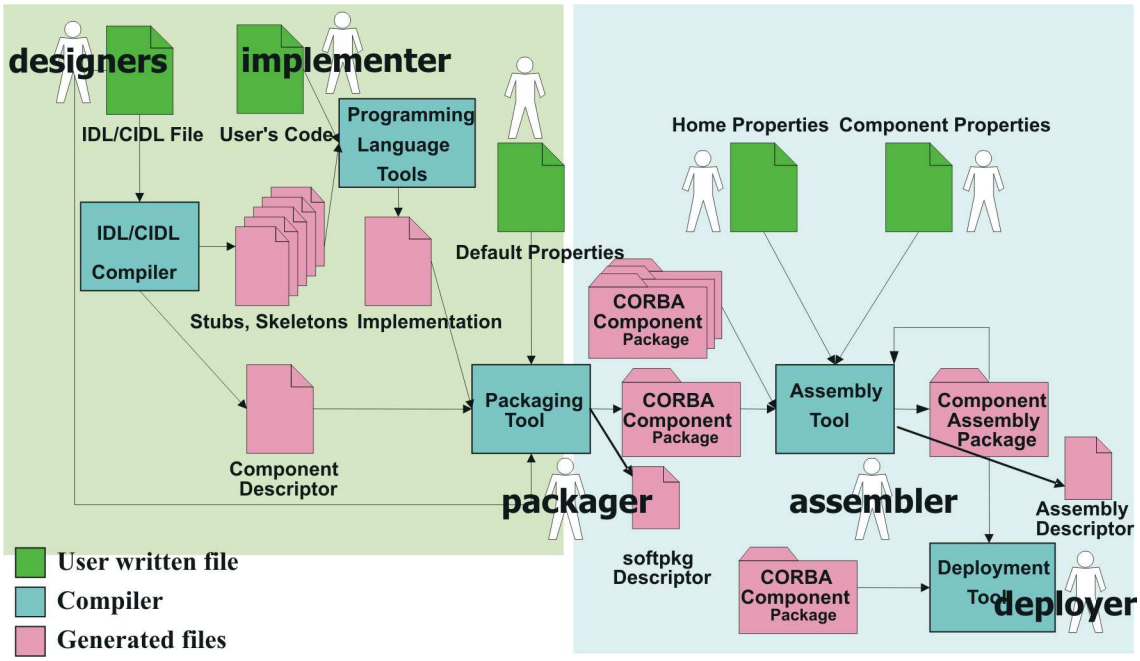
Etap wykonywany przez **implementatora.**

- **Produkowanie paczek** komponentowych w postaci pliku archiwum (*zip*) zawierających:
  — skompilowane klasy komponentów,
  — deskryptory XML komponentów,
  — domyślne własności deskryptorów XML.

  Etap wykonywany przez osobę osobę pełniącą rolę **pakującego (*packager*)** za pomocą interaktywnego narzędzia. Istnieją przypadki, w których etap ten kończy proces tworzenia aplikacji - aplikacja możliwa jest już do osadzenia.
- **Asemblacja komponentów** - wytwarzanie paczek asemblacji w postaci pliku archiwum:
  — paczki z komponentami dopasowane do wymagań stawianych aplikacji,
  — deskryptory asemblacji w postaci plików XML.
  — opis instancji komponentów i połączeń pomiędzy nimi,
  — podział logiczny komponentów.

  Proces może być iteracyjny. Jest wykonywany przez osobę pełniącą rolę **asemblującego (*assembler*)**. Wskazane jet użycie interaktywnego narzędzia wizualizującego tworzoną aplikację. Celem aplikacji tworzonej wraz z tym dokumentem jest właśnie ułatwienie przeprowadzania etapu asemblacji.
- **Alokacja komponentów** - uruchomienie przygotowanych wcześniej archiwów komponentów i archiwów asemblacji. W tej fazie następuje:
  — przypisanie wirtualnych lokalizacji komponentów do fizycznych węzłów,
  — uruchomienie procesu osadzania komponentów - instalacja komponentów w konkretnych węzłach w sieci,
  — na końcu procesu dostajemy zainicjalizowane i skonfigurowane komponenty osadzone w kontenerach CCM.

Cały proces tworzenia aplikacji środowiska *CCM* prezentuje rysunek 2.1.



Rysunek 2.1. Tworzenie aplikacji *CCM*

Najważniejszym dla nas etapem z punktu widzenia tworzenia aplikacji jest faza asemblacji. Powstałe w tej fazie archiwum plików zawiera m.in. zestaw deskryptorów opisujących tworzoną aplikację. Archiwum to posiada następujące cechy:

- może być poddawane stałym modyfikacjom w celu dostosowania implementowanego modelu do potrzeb użytkownika,
- jest samo opisujące się,
- niezależne od zewnętrznych plików.

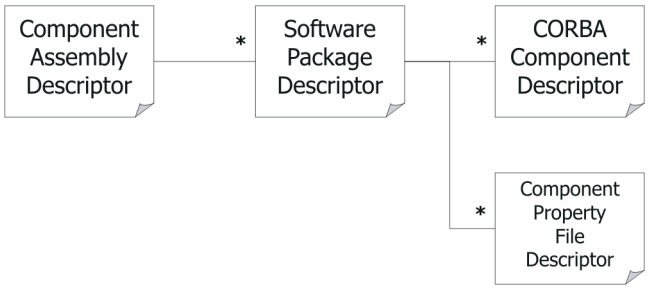
Uruchomienie rozproszonej aplikacji odbywa się za pomocą jednego polecenia. Proces ten jest na tyle łatwy, że nie wymaga od obsługującego doświadczenia w programowaniu.

Istnieje wiele plików deskryptorów potrzebnych do prawidłowego funkcjonowania aplikacji CCM:

- *Software Package Descriptor (.csd).*
  - — opisuje zawartość paczki komponentu,
  - — wskazuje plik implementacji (może być więcej niż jeden).
- *CORBA Component Descriptor (.ccd).*
  - — zawiera techniczne informacje generowane głównie przez kompilator *CIDL (Component Implementation Definition Language)*,
  - — zawiera polityki wybranych kontenerów - ustawiane przez użytkownika.
- *Component Assembly Descriptor (.cad).*
  - — opisuje początkową konfigurację:
    - — *obiektów home* - instancje *obiektów home* do stworzenia,
    - — komponentów - instancji komponentów do stworzenia,
    - — połączeń - połączenia pomiędzy portami komponentów.

- *Component Property File Descriptor (.cpf).*
  — zawiera ustawienia *obiektów home* i komponentów,
  — własności atrybutów konfiguracyjnych w postaci nazwa-wartość.

Zależności pomiędzy poszczególnymi deskryptorami przedstawia rysunek 2.2.



Rysunek 2.2. Relacje pomiędzy poszczególnymi deskryptorami

Wszystkie pliki potrzebne do uruchomienia aplikacji znajdują się z zbiorze asemblacji (plik o rozszerzeniu *.aar*). Uruchomienie rozproszonej aplikacji CCM wymaga podania w parametrze wyłącznie pliku archiwum .aar.

2.1.3. CCM a inne technologie komponentowe (EJB, DCOM, .NET)

CCM nie jest jedyną technologią komponentową dostępną w świecie oprogramowania. Istnieje wiele podobieństw omawianego rozwiązania z rozwiązaniami konkurencyjnymi. CCM zapewnia jednak kilka unikalnych własności.

Własności analogiczne do innych technologi komponentowych zebraliśmy poniżej:

1. SUN Microsystems - Enterprise Java Beans (EJB).
   a) komponenty CORBA są tworzone i zarządzane przez *obiekty home,*
   b) komponenty CORBA wykonywane są w kontenerach wzbogacających je o systemowe usługi zgodnie z deklaratywnym opisem,
   c) uruchamiane w obrębie serwerów komponentowych aplikacji.
2. Microsoft - Component Object Model (COM).
   a) możliwość wykorzystania wejściowych i wyjściowych interfejsów,
   b) wykonywanie synchronicznych operacji i przesyłanie asynchronicznych komunikatów,
   c) zapewnia introspekcję.
3. Microsoft - .NET Framework.
   a) możliwość wykorzystanie różnych języków programowania,
   b) komponenty są pakowane i przesyłane do dystrybucji.

Unikalne własności CCM:

- aplikacje CCM są rzeczywiście rozproszone - mogą być osadzone i uruchomione na wielu węzłach jednocześnie,
- komponenty CORBA mogą być podzielone na wiele klas.

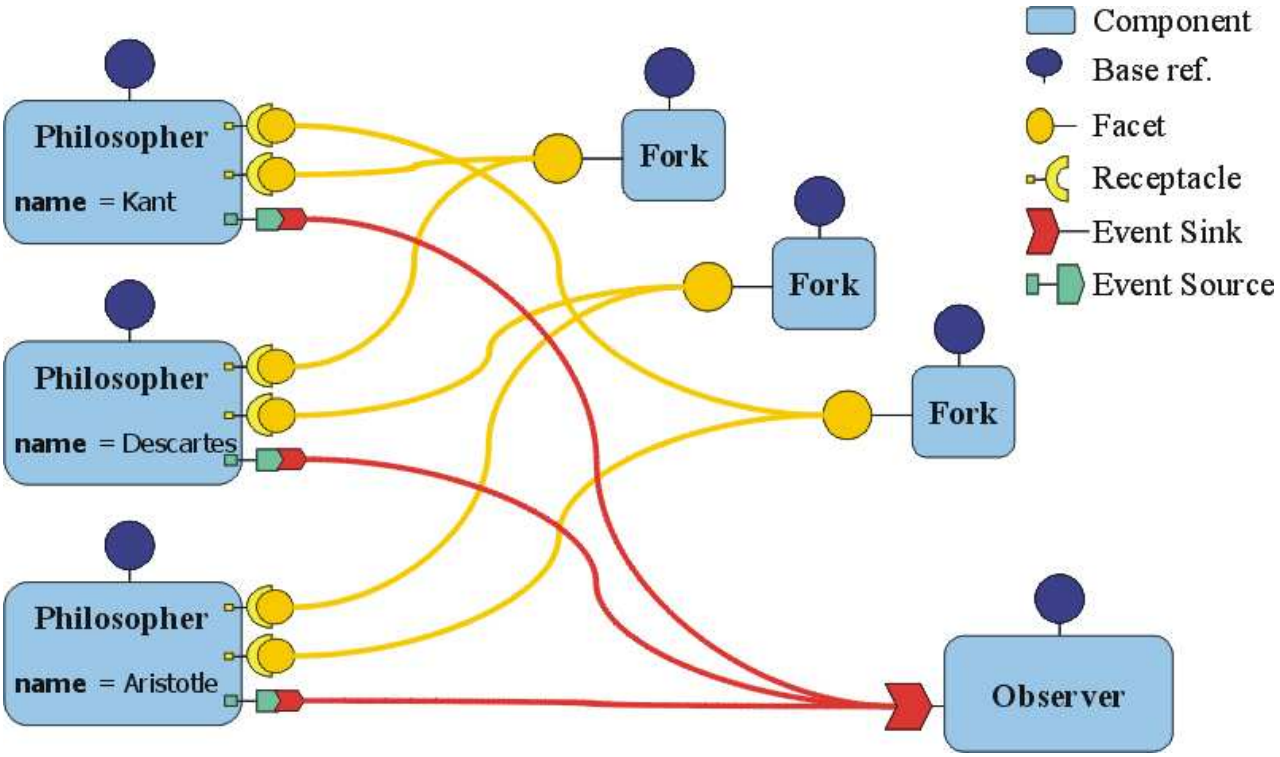
2.1.4. Przykładowa aplikacja w CCM

W środowisku CCM można tworzyć różnego rodzaju aplikacje. Za przykład może posłużyć nam model problemu jedzących filozofów. Problem jest zdefiniowany następująco:

Mamy siedzących przy stole filozofów. Na stole znajduje się tyle samo widelców co jedzących przy stole. Filozof może znajdować się w następujących stanach:

- myśli - tryb czekania,
- jest głodny - próbuje zacząć jeść,
- spożywa posiłek,
- umiera - w wyniku zbyt długiego braku jedzenia.

Aby filozof mógł rozpocząć konsumpcję musi mieć dwa widelce - jeden po prawej i jeden po jego lewej stronie. Model tego problemu w środowisku *CCM* zaprezentowany jest na rysunku 2.3.



Rysunek 2.3. Aplikacja "jedzący filozofowie" w *CCM*

W modelu tym użyto trzech rodzajów komponentów:

- filozof,
- widelec,
- obserwator.

Każdy z filozofów ma możliwość uchwycenia dwóch widelcy - modelowane jest to jako port typu *receptacle* po stronie filozofa oraz jako *facet* po stronie komponentu widelca.

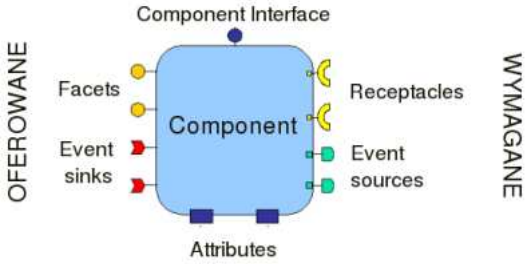
Połączenie to jest synchroniczne. Do połączenia pomiędzy filozofem, a obserwatorem używa się połączenie asynchronicznego. Komponent filozofa posiada port typu *event source* wysyłający komunikaty do portu obserwatora typu *event sink*. Znaczenia portów i ich typów zostaną przedstawione w kolejnym punkcie.

2.2. Elementy CCM użyte w edytorze

Podpunkt przedstawia pokrótce elementy CORBA Component Model jakie zostały użyte przy tworzeniu naszej aplikacji. Celem poniższego opisu jest jedynie przybliżenie czytelnikowi kluczowych pojęć związanych z modelem CCM, pełne zrozumienie środowiska wymaga przewertowania specyfikacji Object Management Group, o których piszemy również w bibliografii.

2.2.1. Komponenty

Komponent jest nowym metatypem stanowiącym kolejny, wyższy poziom abstrakcji po dawno już przyjętym i silnie eksploatowanym w programowaniu obiektowym pojęciu klasy. Podobnie jak w przypadku klas w języku Java tak i też pomiędzy komponentami może występować jednokrotne dziedziczenie (tzn. komponent może wystąpić w relacji generalizacji - specjalizacji tylko z jednym komponentem) oraz komponenty mogą realizować wiele interfejsów. Instancja komponentu posiada stan, zapisany w wartościach atrybutów komponentu. Zachowanie komponentu określone jest poprzez jego porty (zagadnienie portów opisujemy w 2.2.2).
Komponent jest zarządzany przez *obiekt Home* (kolejny metatyp wprowadzony w modelu CORBA Component Model), który odpowiada za tworzenie i likwidacje obiektu komponentu czyli określa cykl życia instancji komponentu.



Rysunek 2.4. Graficzna reprezentacja komponentu (za Philippe Merle, patrz bibliografia [1])

2.2.2. Porty

Port jest to sposób w jaki komponent eksponuje funkcjonalność jaką zapewnia albo taką której wymaga od innych komponentów. CORBA Component Model specyfikuje następujące rodzaje portów:

facet określa interfejs usług jakie komponent zapewnia swoim klientom (innym komponentom),

receptacle określa zestaw wymaganych operacji, komponent poprzez ten typ portu deleguje wykonanie zadania do innych komponentów zapewniających pożądaną funkcjonalność, przeznaczony do komunikacji synchronicznej, receptacle może wskazywać na pojedyńczą instancję jak i również listę instancji komponentów gwarantujących wymagany przez port zestaw usług (mówi się wówcza o porcie *receptacle* typu *multiple*),

event source określa producenta zdarzeń danego typu, wyróżnia się dwa rodzaje źródeł: *emitter* (zdarzenia są konsumowane tylko przez jednego słuchacza) oraz *publisher* (zdarzenia mogą być odbierane przez wielu konsumentów),

event sink określa konsumenta zdarzeń określonego typu, może śledzić zdarzenia produkowane zarówno przez porty typu *emitter* jak i *publisher.*

Kompatybilność portów

Z powyższego zestawienia wyróżnia się dwa zestawy portów komplementarnych:

- facet i receptacle (model komunikacji synchronicznej),
- event source i event sink (model komunikacji *publish/subscribe*).

Na łączenie portów ponadto nałożone są następujące reguły:

1. Aby połączyć dwa porty synchroniczne muszą mieć one ten sam interfejs lub ich interfejsy powinny być w relacji dziedziczenia takiej, że poziom usług gwarantowanych przez *facet* jest wystarczający dla portu *receptacle.*
2. Aby połączyć dwa porty asynchroniczne zdarzenia przez nie generowane muszą być tego samego typu.
3. Do portu *emitter* może być podłączony tylko jeden *event sink.*
4. Do zwykłego *portu receptacle* (delegującego wykonanie usługi do pojedyńczego komponentu) można podłączyć tylko jeden *facet.*

2.2.3. Serwery komponentów

Podobnie jak ma to miejsce w J2EE (Java 2, Enterprise Edition) środowiskiem życia komponentu jest kontener stanowiący swoisty interfejs pomiędzy komponentami a mechanizmami niskiego poziomu, obsługi wielowątkowości, transakcyjności, zarządzania pulami zasobów, zarządzania stanem. Warto zaznaczyć, że kontener w modelu CCM potrafi ściągnąć definincje komponentu z repozytorium w dowolnym momencie, w szczególności dopiero gdy zajdzie potrzeba inicjalizacji instacji komponentu.
Za tworzenie kontenera odpowiadają serwery komponentowe, ich najważniejszą rolą jest inicjalizacja kontenera, w którym następnie osadzany jest *obiekt home* komponentu i w końcu *home* tworzy obiekty komponentów danego typu. Serwer komponentowy mapuje się w systemie na proces zawierający dowolną liczbę kontenerów.
Ponadto w używanej przez nas implementacji modelu *CCM, OpenCCM* wprowadza jako miejsce wykonania jeszcze jeden typ: *node* (*węzeł*). *Node* nie jest elementem specyfikacji *CCM*, z braku dokumentacji zdani byliśmy jedynie na testy empiryczne, z których wysnuliśmy następujące wnioski:

- *node* mapuje się na pojedynczy proces,
- *node* jest agregatem serwerów komponentowych,
- po wskazaniu danego obiektu *node* jako miejsca wykonania komponentu tworzony jest dedykowany dla niego komponent serwer, w nim uruchamiany kontener, w którym alokowana zostaje instancja *home'a* komponentu.

2.2.4. Serwisy

CORBA dostarcza szereg serwisów (m.in *Naming Service*, *Interface Repository*, *Concurrency Service, Transaction Service, Notification Service*) ułatwiających tworzenie rozproszonych aplikacji, w naszej pracy użyteczne były dwa z nich:

Naming Service
Serwis gromadzi skojarzenia typu: *nazwa - obiekt* . Ta asocjacja zawsze jest tworzona w pewnym kontekście stanowiącym zestaw par *nazwa - obiekt,* w których nazwy są unikalne. Jeden obiekt może występować w wielu skojarzeniach w tym samym lub różnych kontekstach.
Ponieważ kontekst traktowany jest jak każdy inny obiekt może zostać skojarzony z nazwą w pewnym kontekście, dzięki tej własności *Naming Service* ma strukturę grafową. Podstawowe usługi zapewniane przez serwis to:
- stworzenie skojarzenia *nazwa - obiekt*,
- reasocjacja obiektu (zmiana nazwy),
- uzyskanie obiektu o podanej nazwie w określonym kontekście,
- wylistowanie grafu asocjacji.

Stworzona przez nas aplikacja korzysta z *Naming Service'u* w celu wykrycia istniejących serwerów komponentów oraz węzłów, jako przyszłych miejsc alokacji i wykonania komponentów.

Interface Repository
Serwis zapewniający informacje na temat interfejsów określonych przy pomocy *OMG IDL'a* (Interface Definition Language), pozwala na introspekcje, tzn aplikacja korzystając z serwisu może uzyskać definicje interfejsu, o którym nie wiedziano na etapie kompilacji. Wykorzystywany jest do:
- sprawdzenia nadchodzących żądań pod względem zgodności typu,
- sprawdzenia poprawności grafu dziedziczenia interfejsów,
- zapewnienia współpracy różnych implementacji *ORB* (*Object Request Broker*).

W *Interface Repository* znajdują się opisy interfejsów pogrupowane w jednostki zwane *modułami. Moduł* może zawierać stałe, definicje typów, interfejsów, komponentów, home'ów komponentów, wyjątków oraz innych modułów (repozytorium ma więc, podobnie jak *Naming Service*, strukturę drzewiastą).
W naszym edytorze *Interface Repository* pełni kluczową rolę. Używamy tego serwisu do pobrania definicji komponentów oraz do walidacji tworzonych pomiędzy portami komponentów połączeń.

2.3. Przegląd narzędzi używanych w OpenCCM

Środowisko *OpenCCM* ułatwia tworzenie aplikacji komponentowych poprzez dostarczenie pokaźnego zestawu skryptów i narzędzi automatyzujących czynności towarzyszące temu procesowi. Lista przez nas używanych programów przedstawia tabela 2.1. Poniżej prezentujemy opis nowych pojęć występujących w tabeli.

OpenCCM Configuration Repository zawiera *IOR* (*Interoperable Object Reference*), *PIDs* (*Process IDentifier*), oraz wyjście *utworzonych OpenCCM* procesów.

OpenCCM Distributed Computing Infrastructure Manager zarządza domeną połączonych węzłów tworzących zunifkowane środowisko wykonywania aplikacji komponentowych.

OpenCCM Comanche Server mikro serwer HTTP oraz multicast serwer zapewniający zdalny dostęp do *OpenCCM Configuration Repository.*

2.4. Deskryptory asemblacji CAD

Jak wspomnieliśmy w punkcie 2.1.2 model *CCM* korzysta z wielu deskryptorów. Z punktu widzenia stworzonej przez nas aplikacji najważniejszym plikiem tego typu jest deskryptor *.cad (Component Assembly Descriptor).*

2.4.1. Funkcja pliku CAD

Plik *.cad* pozwala na ustawienie własności rozproszonej aplikacji *CCM*. Deskryptor ten pozwala na określenie wykorzystywanych paczek komponentów, przyporządkowanie *obiektów home* komponentów do fizycznych węzłów oraz określenie połączeń pomiędzy komponentami.

2.4.2. Opis sekcji pliku CAD

Deskryptor asemblacji składa się z następujących sekcji:

componentfiles - sekcja ta zawiera pliki komponentów (*Component ARchive* - plik o rozszerzeniu *.car*) używanych w zbiorze *.aar*. Przynajmniej jeden komponent musi być zdefiniowany. Sekcja ta nadaje unikalne identyfikatory każdemu z plików komponentów, które to identyfikatory stają się referencjami do komponentów w innych sekcjach pliku *.cad,*

partitioning - sekcja ta przyporządkowuje *obiekty home* i komponenty do procesów i hostów. Poszczególne użycia komponentu są związane z *obiektem home* danego komponentu. *Home* wraz z jego komponentami opisywany jest w podsekcjach *homeplacement,*

connections - sekcja ta służy do określenia połączeń pomiędzy komponentami. Operacje synchroniczne opisywane są w podsekcji *connectinterface*. Element *connectevent* natomiast opisuje połączenia asynchroniczne pomiędzy komponentami.

nazwa	**funkcja**
ccm_ install	instaluje *OpenCCM Configuration Repository*
ccm_ deinstall	deinstaluje *OpenCCM Configuration Repository*
ir3_ start	uruchamia *Interface Repository*
ir3_ stop	wyłącza *Interface Repository*
ir3_ feed	*wprowadza defincje* opisane z pomocą *IDL'a* do *repozytorium*
ir3_ set	ustawia *Interface Repository* na podane przez użytkownika
ns_ start	uruchamia *Naming Service*
ns_ stop	wyłącza *Naming Service*
ns_ set	ustawia *Naming Service* na podany przez użytkownika
dci_ start	uruchamia *OpenCCM Distributed Computing Infrastructure Manager*
dci_ stop	wyłącza *OpenCCM Distributed Computing Infrastructure Manager*
factory_ start	uruchamia *OpenCCM Assembly Factory Manager*
factory_ stop	wyłącza *OpenCCM Assembly Factory Manager*
jcs_ start	uruchamia serwer komponentów
jcs_ stop	wyłącza serwer komponentów
node_ start	uruchamia node'a
node_ stop	wyłącza node'a
ccm_ deploy	uruchamia aplikacje komponentową
tear_ down	wyłącza aplikację komponentową
ccm_ explorer	narzędzie do zarządzania komponentami na etapie uruchomienia
comanche_ start	uruchamia *OpenCCM Comanche Server*
comanche_ stop	wyłącza *OpenCCM Comanche Server*

Tablica 2.1. Zestawienie używanych przez nas skryptów i narzędzi zawartych w *OpenCCM*

Pełny wykaz elementów pliku *CAD* zawarty jest w dokumencie *OMG*, formal/02-06-65 (patrz bibliografia [2]), zaś wykorzystywany przez nas podzbiór elementów zamieszczamy w *Dodatku A.*

2.4.3. Alternatywny sposób asemblacji

Proces asemblacji i alokacji komponentów jest kluczowym elementem budowania aplikacji rozproszonej. Procesy te można wykonać w *OpenCCM-ie* na dwa możliwe sposoby:

- imperatywny - wykrzystanie plików źródłowych i skryptu uruchomieniowego *start_java,*
- deklaratywny - w oparciu o deskrytpory **.csd* i **.cad.*

O ile sposób imperatywny dzięki dostępnym przykładom nie stanowił większego problemu, o tyle druga metoda okazuje się żmudna i wymagająca niemałej wprawy i wiedzy na temat konfiguracji procesu asemblacji i alokacji. Sposób imperatywny nie może stanowić satysfakcjonującego rozwiązania w dynamicznie rozwijanym środowisku komponentowym, gdyż wymaga dostępu do kodu i jego wtórnej kompilacji. Wyniesione z tego przykładu doświadczenie utwierdza nas w przekonaniu o potrzebie zautomatyzowania procesu ponownej asemblacji i uruchamiania oraz stworzenia bardziej przyjaznego dla użytkownika interfejsu pozwalającego na efektywną pracę w oparciu o technologię *CCM.*

2.4.4. Proces asemblacji w specyfikacji *CCM 4.0*

W połowie roku 2006 pojawiła się nowa specyfikacja modelu *CCM* - wersja 4.0 [18]. Pomimo tego, że praca ta bazuje na specyfikacji wcześniejszej - wersji 3.0, uznaliśmy, że warto przedstawić podstawy nowego sposobu asemblacji wprowadzonej wraz z modelem *CCM 4.0,* co też czynimy w tej sekcji.

Nowa specyfikacja używa innych schematów plików *XML.* Pewne elementy schematu zostały niezmienione, lecz pozostała część (przede wszystkim znaczniki podstawowe) uległa znacznym modyfikacjom. Ocenia się, że w przypadku typowego użycia specyfikacji w wersji 3.0 proces asemblacji będzie zgodny w przód z tym opisywanym w specyfikacji modelu *CCM 4.0* oraz, że istnieje możliwość zbudowania automatu dokonującego translacji pomiędzy tymi standardami.

Ze względu na trudności w stworzeniu funkcjonalnych narzędzi do wspomagania procesu asemblacji specyfikowanego przez standard 3.0, w nowej wersji specyfikacji zdecydowano się na uproszczenie struktury plików umożliwiając w ten sposób budowę stosunkowo prostych aplikacji automatyzujących ten proces.

Nowa specyfikacja rezygnuje ze wszystkich tych elementów, które nie były bezpośrednio związane z procesem osadzania i konfiguracji. Ponadto, standard 3.0 nie definiował pewnych kwestii związanych z informowaniem kontenera o wymaganiach instancji komponentów. Kwestie te musiały być rozwiązywane indywidualnie przez poszczególnych producentów implementacji modelu *CCM.* Dlatego też model *CCM 4.0* definiuje dwa kanały komunikacyjne służące wymianie informacji pomiędzy aplikacjami generującymi kod oraz środowiskami uruchomieniowymi. Twórcy specyfikacji wierzą, że

dzięki takiemu podejściu nastąpi niezależny rozwój funkcjonalnych środowisk specjalizowanych w tworzeniu aplikacji oraz środowisk zajmujących się kwestiami uruchomieniowymi przygotowanej wcześniej aplikacji rozproszonej.

Ze względu na duże zmiany specyfikacji wersji 4.0 w stosunku do 3.0 narzędzia wspomagające osadzanie aplikacji rozproszonych muszą być przebudowywane, choć jest możliwe powtórne użycie elementów narzędzi już istniejących.

2.5. Podsumowanie

CCM dostarcza bogatą funkcjonalność przydatną w konstruowaniu nawet znacznie rozbudowanych aplikacji rozproszonych. Model ten jednak jest bardzo złożony. Automatyczne generowanie pliku *.cad* za pomocą edytora wizualnego przynajmniej częściowo automatyzuje proces budowania rozproszonej aplikacji w *CCM*. Jak widać plik .cad to tylko jeden z deskryptorów, a więc do pełnego zautomatyzowania procesu tworzenia aplikacji *CCM* jeszcze daleko.

Rozdział 3

Wybór narzędzia do budowy interfejsu graficznego - GEF

Podczas projektowania aplikacji generującej deskryptor *.cad* rozważaliśmy użycie różnych gotowych narzędzi do generowania plików *XML* z modelu graficznego. Badaliśmy możliwości istniejących środowisk, które możnaby zastosować do graficznej reprezentacji komponentów *CCM* oceniając jednocześnie ich przydatność do zastosowania w naszym projekcie.

3.1. Wybór środowiska wspomagającego tworzenie diagramów

Najważniejszymi kryteriami decydującymi o przydatności danego rozwiązania do zastosowania w naszym projekcie były:

- czytelna, graficzna reprezentacja komponentów oraz połączeń pomiędzy nimi,
- interakcyjność - łatwość modyfikacji tworzonego modelu za pomocą myszki i klawiatury,
- możliwość zapisywania i wprowadzania zmian do tworzonego modelu,
- możliwość współpracy z repozytorium interfejsów (*Interface Repository*) oraz serwisem nazw (*Name Service*),
- łatwość dostosowywania rysowanych obiektów do potrzeb edytora,
- eksport modelu do pliku typu .xml,
- wieloplatformowość,
- rozwiązanie bezpłatne z możliwością modyfikacji kodu (*open source*).

Przetestowaliśmy kilka środowisk pod kątem przydatności w realizowanym projekcie. Poniższe punkty prezentują sprawdzane środowiska odnosząc je do kryteriów przedstawionych powyżej.

3.1.1. *EMF* w połączeniu z *UML2*

EMF (Eclipse Modeling Framework) jest narzędziem do modelowania i generowania kodu aplikacji z modelu opisanego w pliku *XMI (XML Metadata Interchange)*. *EMF* posiada funkcjonalność wygenerowania klas *Java* z modelu; posiada zbiór adapterów, które umożliwiają przeglądanie i edytowanie modelu oraz sam edytor modelu. Model może być wyspecyfikowany za pomocą:

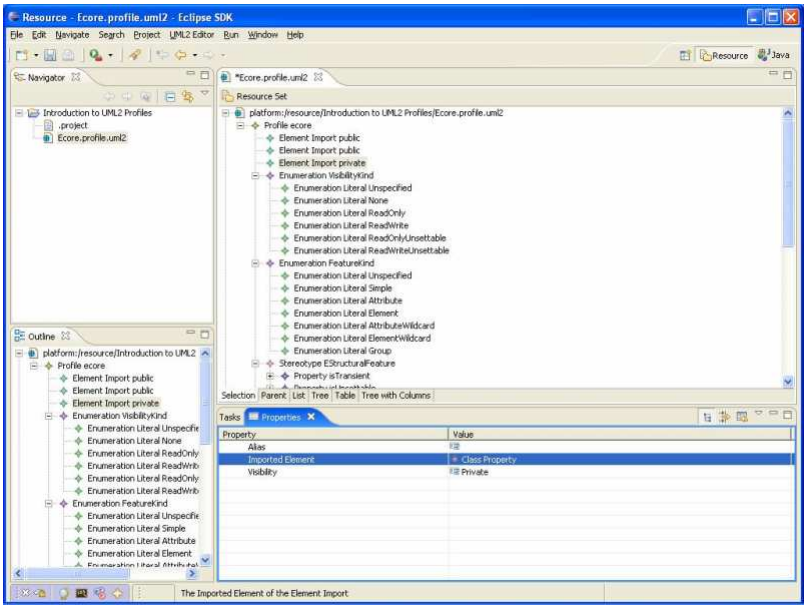
- annotated Java,
- pliku *XML*,
- narzędzi typu Rational Rose.

Model oprócz trzech rodzajów reprezentacji przedstawionych powyżej jest serializowany do pliku *XMI* (*XML Metadata Interchange*). Jest to bardzo pożądana funkcjonalność, ponieważ po zdefiniowaniu modelu otrzymujemy przyjazną postać *XML*. Plik *XMI* należałoby w następnym kroku przekonwertować do standardu pliku *.cad*. Ze względu na popularność języka *XML* i narzędzi do jego obsługi konwersja taka wydaje się być etapem łatwym.

UML2 jest implementacją metamodelu *UML 2.0* dla platformy *Eclipse* bazującą na *EMF*. Łącząc środowisko *EMF* z *UML2* uzyskujemy możliwość modelowania komponentów, które specyfikuje standard *UML 2.0*.

Zalety użycia *EMF* w połączeniu z *UML2*:
- pracuje jako wtyczka do stabilnego środowiska jakim jest *Eclipse*,
- posiada zaimplementowane tworzenie graficznych reprezentacji komponentów,
- umożliwia łączenie jedynie kompatybilnych komponentów, co rozwiązuje problem semantycznej walidacji połączeń.

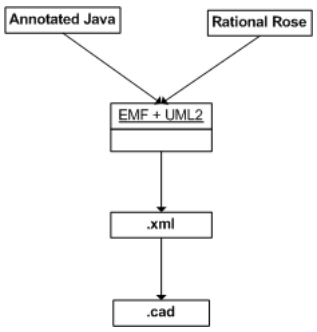
Wady użycia *EMF* w połączeniu z *UML2*:
- prezentuje model jako drzewo, nie jako diagram,
- stosuje nietypowy sposób wczytywania modelu - model może być wczytywany z *annotated Java*, pliku *XML*, albo za pomocą narzędzi typu *Rational Rose,*
- wymaga korzystania ze środowiska *Eclipse*,
- generuje plik *.xml*, pozostaje otwarta kwestia konwersji do formatu pliku *.cad*,
- znajduje sie wciąż w trakcie rozwoju: standard *UML2* jest jeszcze na etapie krystalizacji,
- nie posiada kompletnej dokumentacji *UML2*.



Rysunek 3.1. Przykładowe drzewo stworzone w EMF

Reprezentacja modelu w postaci drzewa skreśla możliwość skorzystania z niewątpliwie dobrej jakości środowiska, jakim jest *EMF*. Najbardziej intuicyjną i przyjazną użytkownikowi reprezentacją byłoby operowanie bezpośrednio na diagramach wzorowanych na *UML*. *EMF* nie udostępnia jednak takiej funkcjonalności. Dlatego też zrezygnowaliśmy z tego rozwiązania.

Możliwość wykorzystania *EMF* przedstawiliśmy na schemacie przedstawionym na rysunku 3.2.



Rysunek 3.2. Możliwość wykorzystania EMF

3.1.2. Narzędzia do modelowania w *UML2.0* na przykładzie *StarUML*

Dostępnych jest wiele narzędzi umożliwiających modelowanie w *UML2.0*. Jednak większość dobrych narzędzi *UML2.0* jest płatnych albo posiada skromną funkcjonalność. Narzędzia te zwykle mają inne przeznaczenie niż cel, do którego chcielibyśmy

je użyć. Spośród oprogramowania do modelowania za pomocą *UML2.0* najlepiej prezentował się *StarUML*.

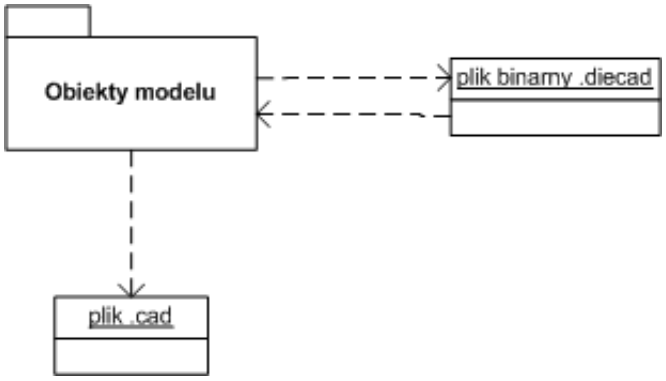
Zalety:
- udostępniony na zasadach otwartego oprogramowania (*open source*),
- wspiera *UML2.0*,
- umożliwia zapis modelu do pliku *XML*.

Wady:
- bywa niestabilny,
- nie wspiera reprezentacji portów - wymagane zmiany w kodzie źródłowym,
- działa tylko w środowisku *Windows*.

Wymienione wady są na tyle poważne, że użycie tego narzędzia stało się niemożliwe.

3.1.3. *GEF*

GEF (*Graphical Editing Framework*) jest rozbudowaną biblioteką służącą do graficznej reprezentacji modelu. Występuje jako wtyczka do środowiska *Eclipse*. Od początku testów *GEF* zapowiadał się jako dobre środowisko mogące znacznie ułatwić realizację budowy graficznego edytora do generowania deskryptora *.cad*. Biblioteka ta wspiera zapis modelu do pliku binarnego, lecz nie zawiera wbudowanego zapisu modelu do pliku typu *XML*. Tak więc ta funkcjonalność musi być dodana przez nas. Sposób zapisu i odczytu do i z plików przedstawia rysunek 3.3.



Rysunek 3.3. Zapis do plików z biblioteki *GEF*

Wyniki testowania tego środowiska przedstawione są poniżej:

Zalety:
- możliwość realizacji dowolnego modelu zawierającego obiekty i połączenia między nimi,
- obsługa akcji użytkownika,
- możliwość zapisu modelu w postaci binarnej,
- łatwa rozbudowa o dowolną funkcjonalność użyteczną np. do współpracy do repozytorium interfejsów czy też serwisu nazw,

- współpraca z wieloma systemami operacyjnymi - środowisko napisane w przenośnym języku *java*,
- oprogramowanie otwarte (*open source*).

Wady:
- integralna część środowiska *Eclipse,*
- brak wbudowanego eksportu do pliku typu *XML*.

3.1.4. Inne testowane narzędzia

Część testowanych środowisk posiadała na tyle poważne wady, że nawet bez dogłębnych testów wiadomo było, że skorzystanie z nich nie będzie możliwe. Następujące punkty przedstawiają te aplikacje:

1. ***CoSMIC***
   Component Synthesis with Model Integrated Computing (CoSMIC) to zestaw narzędzi do modelowania. Dokładne możliwości nie zostały do końca przeanalizowane, ponieważ posiada poważne wady, które dyskwalifikują możliwość użycia tego środowiska do naszych potrzeb:
   - dostępny tylko na platformę *Windows*,
   - jeszcze nie ma wersji 1.0,
   - problem z przygotowaniem struktury - nie ma odwołań do *InterfaceRepository.*
2. ***LabView***
   LabView (Laboratory Virtual Instrumentation Engineering Workbench) jest środowiskiem do programowania wizualnego. Jest ono szeroko używane przez środowiska badawcze oraz przemysłowe. Narzędzie jest pomocne w graficznym projektowaniu programów dzięki wykorzystaniu szerokiej gamy bibliotek zapewniających funkcjonalność związaną z analizą danych, tworzeniem raportów, pobieraniem danych oraz odczytem i zapisem plików. Niestety narzędzie to jest płatne (kilka tysięcy złotych), co powoduje, że nie byliśmy w stanie dogłębnie przetestować tego środowiska i co dyskwalifikuje możliwość użycia tego oprogramowania w naszym projekcie.

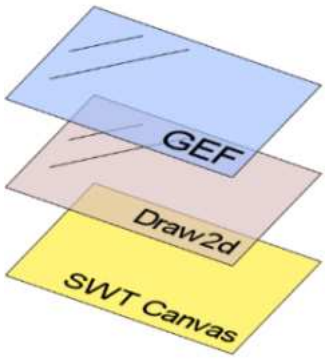
3.1.5. Podsumowanie wyboru środowiska

Spośród testowanych rozwiązań najlepsza okazała się biblioteka do modelowania działająca jako wtyczka do *Eclipse* o nazwie *GEF (Graphical Editing Framework).* Wybrane środowisko spełniało wszystkie nasze oczekiwania poza wbudowanym zapisem modelu do pliku *XML.* Generowanie pliku *.cad* musiało więc zostać zaimplementowane ręcznie. Poza tym *GEF* wymaga do pracy środowiska *Eclipse,* które jest bardzo funkcjonalne, ale jednocześnie wymaga sporej ilości miejsca na dysku (standardowy rozmiar programu przekracza 100MB). *GEF* jest szczegółowo opisany w dalszej części tego rozdziału.

3.2. Wprowadzenie do GEF

Graphical Editing Framework (*GEF*) jest środowiskiem pozwalającym na wizualizacje i edycje niemal każdego modelu, wykorzystującym w tym celu dwie wtyczki (rozszerzenia środowiska *Eclipse*):

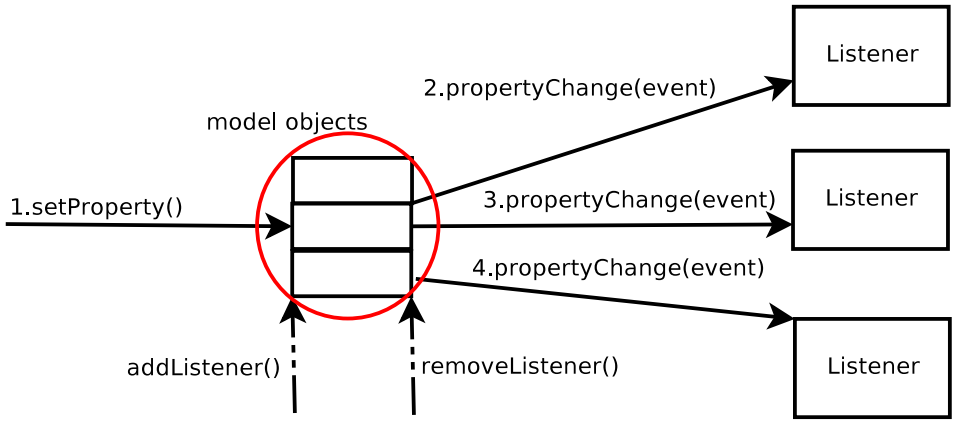
- **org.eclipse.draw2D** - zapewniający graficzną wizualizację reprezentacji modelu poprzez:
  — efektywne tworzenie reprezentacji geometrycznej,
  — dostarczenie licznego zbioru kształtów geometrycznych,
  — obsługę wielu warstw (w tym przeźroczystych),
  — okno podglądu.
  Draw2D jest swego rodzaju nakładką na bibliotekę *Standard Widget Toolkit* (*SWT*), do której delegowane są operacje związane z wizualizacją graficzną.
- **org.eclipse.gef -** zawiera właściwą logikę ułatwiającą tworzenie graficznych edytorów modeli, w tym, m.in:
  — narzędzia do zaznaczania, tworzenia, łączenia elementów modelu (dostępne z poziomu palety edytora),
  — dwie perspektywy widoku modelu (graficzna i drzewiasta),
  — kontroler zajmujący się translacją akcji użytkownika w edytorze na zmiany w modelu.



Rysunek 3.4. Warstwowy model edytora tworzonego z wykorzystaniem biblioteki *GEF* (patrz [15])

Z pomocą biblioteki *GEF* można łatwo i szybko stworzyć edytor zarówno o prostych funkcjach (np. modyfikacja właściwości elementów modelu), jak i bardziej rozbudowanych (np. zmiana struktury modelu), a dodatkowo realizacja ich na różne sposoby nie stanowi większego problemu. Kolejną zaletą środowiska *GEF* jest wsparcie dla znanych i chętnie wykorzystywanych przez użytkowników mechanizmów, takich jak:

- *drag & drop,*
- *copy & paste*,
- *redo & undo*,
- operacje wybierane z rozwijanych menu lub pasków narzędzi.



Rysunek 3.5. Mechanizm notyfikacji

3.3. Tworzenie graficznych edytorów w środowisku GEF

Tworzenie graficznych edytorów w środowisku *GEF* odbywa się w oparciu o architekturę *Model-Widok-Kontroler (ang. Model-View-Controller, MVC).* Pokrótce przyjrzyjmy się głównym założeniom tego wzorca architektonicznego oraz jego implementacji w *GEF.*

3.3.1. Model

Model, czyli uproszczony obraz rzeczywistości, a najczęsciej jakiegoś jej elementu, w rozważanej implementacji *MVC* powinien spełniać następujące warunki:

- *model musi zawierać wszystkie dane przeznaczone do edycji przez użytkownika* - wszystkie dane, które mają być trwale zapisywane należy zawrzeć w tworzonym modelu (dotyczy to również danych opisujących graficzne właściwości widoku modelu, np. wymiary komponentu),
- *model* ***nie zawiera żadnych*** *referencji do widoku czy też pozostałych części edytora* - model stanowić ma jedynie kontener dla edytowalnych danych oraz informować o ich zmianach widok poprzez mechanizm notyfikacji,
- *model musi implementować mechanizm notyfikacji* - mechanizm powinen zapewniać możliwość rejestrowania *słuchaczy (tzw. listener)* i generować zdarzenie przy każdej modyfikacji modelu, jak pokazano na rysunku 3.5.

GEF nie zakłada nic na temat używanego modelu. Dzięki takiemu podejściu można wykorzystywać w tym środowisku niemal każdy model. Z drugiej strony pociąga to za sobą konieczność przestrzegania przedstawionych reguł przez samego projektanta, w szczególności dotyczy to mechanizmu notyfikacji. Dzięki istniejącym już w środowisku *Eclipse* rozwiązaniom nie stanowi to wielkiego utrudnienia.

3.3.2. Widok

Widok stanowi zbiór bloków składających się na graficzny interfejs. Podobnie jak dla modelu architektura *MVC* stawia przed *widokiem* specyficzne wymagania do spełnienia:

- *widok nie może przechowywać żadnych ważnych danych, które uprzednio nie przedstawiono w modelu* - jest to konsekwencja wymagań nałożonych na model,
- *widok* ***nie zawiera żadnych*** *referencji do modelu czy też pozostałych części edytora* - widok jest modułem nie biorącym zupełnie udziału w logice przetwarzania; można go postrzegać jako mapę używaną przez algorytm malujący model do graficznego odwzorowania modelu.

Widok budowany jest w środowisku *GEF* w oparciu o figury z biblioteki *Draw2d,* zawierającej podstawowe kształty geometryczne jak i m.in. możliwość tworzenia figur w oparciu o znane formaty graficzne.

3.3.3. Kontroler

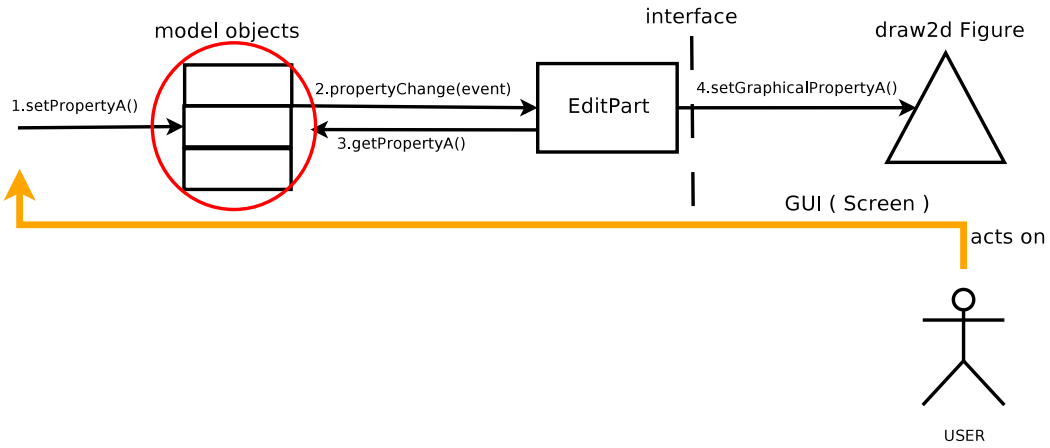
Z wymagań nałożonych na *Model* i *Widok* wynika, że nie są oni wzajemnie świadomi swojego istnienia. Zadaniem kontrolera jest stworzenie połączenia pomiędzy tymi modułami.

W środowisku *GEF'* kontroler tworzy zbiór podklas klasy *EditPart.* Pomiędzy każdym obiektem z *Modelu,* a jego obrazem w *Widoku* jako łącznik występuje obiekt klasy *EditPart.* Obiekt *EditPart* pełni również funkcję *listener'a* dla zdarzeń wywoływanych zmianami w elemencie *Modelu* - w odpowiedzi na nie kontroler modyfikuje w sposób właściwy *obraz w Widoku.* Jak widać *Kontroler* uczestniczy w procesie edycji obiektu z *Modelu.*

3.3.4. Sposób działania

Sposób odwzorowania akcji użytkownika przez *MVC* przedstawia rysunek 3.6.

Dla prostoty pominięto szczegóły implementacyjne określające w jaki sposób akcje wykonywane przez użytkownika na GUI przekładają się na zmiany w modelu (to zagadnienie zostało omówione w podpunkcie 3.4).



Rysunek 3.6. Sposób działania architektury MVC

3.3.5. Zalety wykorzystania architektury Model - Widok - Kontroler

Stosowanie omawianego wzorca architektonicznego niesie za sobą wiele korzyści z których najważniejsze to:

1. *Efektywna dekompozycja funkcjonalna na trzy moduły.*
   Model pozostaje przejrzysty ponieważ cała logika związana z jego edycją znajduje się w *Kontrolerze*, nie posiada żadnych zależności czy referencji do pozostałych modułów - zyskuje się możliwość łatwego ponownego jego wykorzystania w innych aplikacjach (podobnie sytuacja ma się z *Widokiem).*
2. Możliwość łatwej podmiany *Widoku* pod warunkiem wydzielenia interfejsu pomiędzy *Kontrolerem,* a *Widokiem.*
3. Zachowanie przejrzystości i logiki implementacji.
   Dane przechowywane są w *Modelu*, sposób wizualizacji w *Widoku,* a sposób ich powiązania określa *Kontroler.*

3.4. Technika programowania w GEF

Dla lepszego zrozumienia technik programowania w *środowisku GEF* przyjrzyjmy się na początek sposobowi jego działania:

1. Interakcje podejmowane przez użytkownika z edytorem tłumaczone są na *żądania* (*ang. requests*).
2. Kontroler (wspomniane w podpunkcie 3.3.3 obiekty *EditPart*) obsługuje żądania przekazując je do ustalonych *polityk edycyjnych* (tzw. *EditPolices*).
3. *Polityki edycyjne* zamieniają *żądania* na *komendy GEF'a.*
4. *Komendy* zostają wykonane powodując modyfikację edytowanego modelu.

5. Model obserwowany jest przez kontroler: w odpowiedzi na zmianę w modelu kontroler uaktualnia widok.

Wszystkie wymienione elementy zostaną dokładnie omówione w tym podpunkcie.

3.4.1. Struktura edytora

Struktura każdego edytora tworzonego w oparciu o *GEF* zawiera pewne stałe elementy. Na rysunku 3.7 przedstawiono schemat edytora, którego model opisany grafem sklada się z:

- węzłów (*nodes)*,
- połączeń *(connections)*.

GraphEditor

Główna klasa przykładowego edytora. Odpowiedzialna jest za wczytanie i zapis edytowanego modelu oraz stworzenie jego widoku.

EditDomain

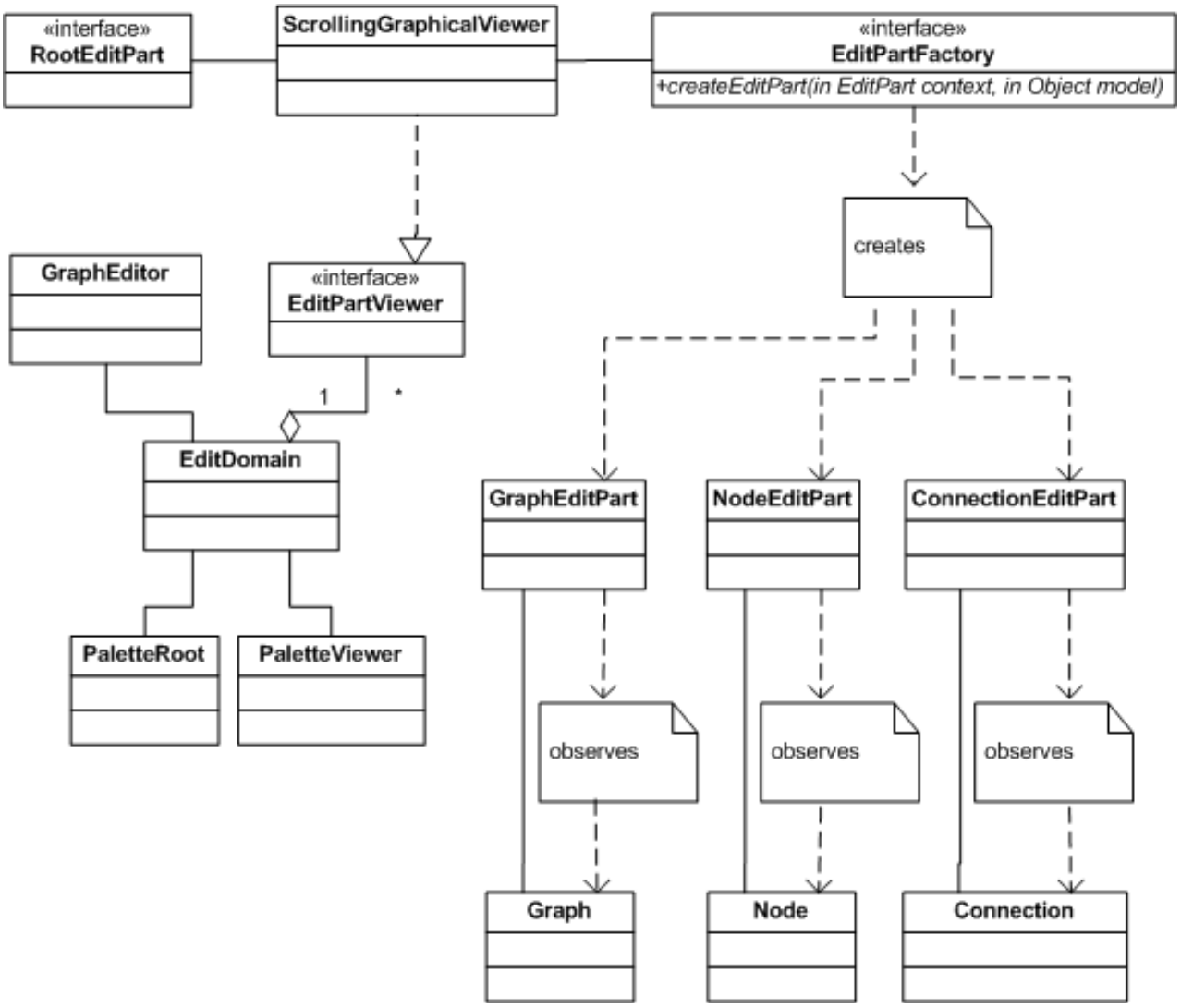
Klasa stanowiąca logiczny interfejs pomiędzy edytorem, widokiem i narzędziami. Zapewnia stos poleceń (*CommandStack),* dzięki czemu możliwe jest w łatwy sposób włączenie takich mechanizmów jak *undo/redo*, a także stwierdzenie czy model uległ modyfikacji i co za tym idzie wymaga zapisania. Zwykle występuje jeden obiekt *EditDomain* na edytor, choć w przypadku edytorów wielostronicowych (tzn. wspierających edycje wielu plików równocześnie w ramach jednej aplikacji) obiekt ten może być współdzielony.

PaletteViewer

Klasa służąca do wizualizacji palety edytora na podstawie *PaletteRoot*.

PaletteRoot

Klasa pełniąca funkcje kontenera dla elementów palety (*PaletteEntry*). Dopuszczalne i zalecane jest organizowanie elementów palety za pomocą *PaletteGroup* oraz *PaletteDrawer* (grupa elementów może zostać *"zwinięta"*).



Rysunek 3.7. Struktura typowego edytora tworzonego z pomocą biblioteki *GEF*

EditPartViewer

EditPartViewer tworzy *wrapper*[1] dla kontrolek zawartych w *SWT*. Klasy implementujące interfejs (np. *ScrollingGraphicalViewer, TreeViewer*) przeprowadzają wizualizacje modelu, u której podstaw leżą kontrolki SWT. Kolejnym ich zadaniem jest zarządzanie rejestrem utworzonych *EditParts.*

RootEditPart

Specjalny rodzaj *EditPart* (tworzący korzeń hierarchi *EditPart*, odwzorowujący się w widoku na tzw. warstwy (*layers*)) nie ma nic wspólnego z korzeniem modelu (w naszym przykładzie korzeń stanowi graf). Zadaniem klas implementujących (np. *ScalableFreeformRootEditPart*) jest stworzenie jednolitego środowiska dla kontrolera - *RootEdit-*

[1] wzorzec projektowy, polegający na tworzeniu klasy/klas dostosowującej interfejs innych klas do potrzeb klienta

Part można zatem traktować jako interfejs pomiędzy *EditParttViewer a EditParts* kontrolujących elementy modelu.

GraphEditPart, NodeEditPart, ConnectionEditPart

Lista *EditParts* stanowiących kontroler dla poszczególnych elementów modelu (zwykle hierarchia *EditParts* odzwierciedla hierarchie klas modelu). EditParts są kluczowym elementem *GEF'a,* określają w jaki sposób model mapuje się na widok oraz jak widok zachowuje się w różnych sytuacjach w czasie edycji.Wyróżnia się trzy rodzaje *EditPart'ów:*

- **GraphicalEditPart** zapewniają graficzną reprezentację elementu modelu.
- **ConnectionEditPart** reprezentują połączenia pomiędzy *GraphicalEditPart.*
- **TreeEditPart** wykorzystywane do tworzenia drzewiastego widoku modelu.

GEF zapewnia domyślną implementacje poszczególnych rodzajów *EditPart (AbstractGraphicalEditPart, AbstractConnectionEditPart, AbstractTreeEditPart*), zadaniem programisty jest przeciążenie paru metod:

createFigure() zwraca figurę (obiekt widoku) reprezentującą obiekt modelu skojarzony z daną instancją *EditPart,*

refreshVisuals() odpowiada za odświeżenie figury widoku w oparciu o aktualny stan skojarzonego z *EditPart* obiektu modelu,

getContentPane() zwraca *content pane* (panel zawartości jest to przestrzeń w której zostanie zwizualizowany obiekt modelu),

getModelChildren() zwraca listę obiektów modelu, które powinny zostać zwizualizowane w obrębie panelu zawartości danego obiektu modelu.

EditPartFactory

Klasy implementujące ten interfejs odpowiedzialne są za tworzenie obiektów *EditPart* dla instancji elementów modelu. Implementacja fabryki *EditPart* polega na realizajci metody:
createEditPart(EditPart context, Object modelElement).

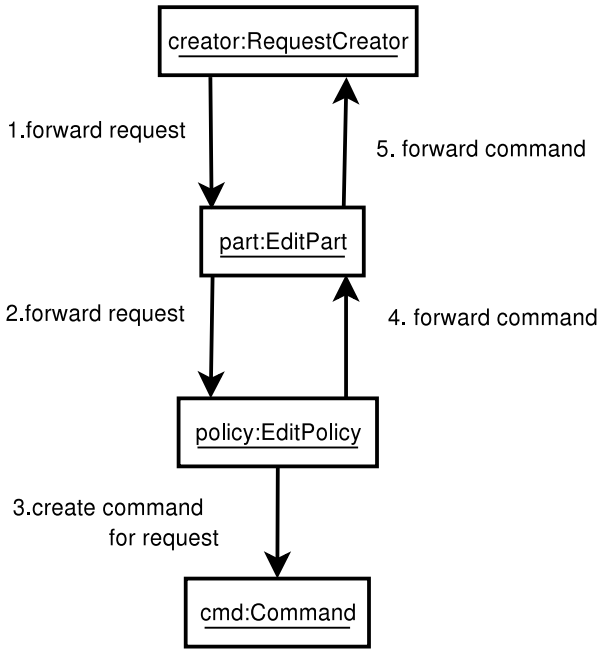
Graph, Node, Connection

Lista klas tworzących model (analogiczna do hierarchii klas kontrolera).

3.4.2. Edycja modelu w środowisku *GEF*

Użytkownik edytora modyfikuje model używając w tym celu, tzw. narzędzi (*tools* - zebranych na paskach narzędzi), akcji (*actions* - wylistowanych w menu), oraz mechanizmu *drag & drop.* Wszystkie te możliwości zostały na rys. 3.8 sklasyfikowane jako obiekt klasy *RequestCreator.* W środowisku GEF żądania dzieli się na trzy klasy:

- *CreateRequests* - używane wszędzie tam gdzie wymagane jest stworzenie nowego obiektu modelu,
- *GroupRequests* - klasa obejmuje żądania dotyczące grupy elementów w odpowiedzi na które wykonywana jest jedna komenda (np. zmiana rozmiaru, przemieszczenie),
- *LocationRequests* - grupuje żądania, dla których stworzone komendy będą się różnić w zależności od miejsca (lokacji) powstania żądania (np. zaznaczenie elementu).



Rysunek 3.8. Zamiana żądania na komendę w środowisku *GEF*

Obiekt kontrolera nie tworzy sam komendy w odpowiedzi na nadchodzące żądanie, ale deleguje to zadanie do obiektu *EditPolicy*. To właśnie polityki edycyjne stanowią o sile środowiska *GEF*, bez nich element kontrolera (obiekt *EditPart)* nie jest w stanie zmodyfikować modelu. Polityki edycyjne są podzielone na *role* - dany *EditPart* może posiadać tylko jedną *EditPolicy* dla konkretnej roli. Najczęściej stosowane role w środowisku *GEF* to:

Component role podstawowa rola, wykorzystywana do realizacji żądań operujących wyłącznie na obiekcie modelu i nie wymagających przeprowadzenia żadnych dodatkowych akcji na edytorze,

Connection role ma to samo zastosowanie co *Component role,* ale dotyczy połączeń pomiędzy obiektami z modelu,

Container role rola odpowiedzialna za tworzenie komend wykonywanych na kontenerze (np. tworzenie obiektów potomnych, czyli obiektów zawartych w danym elemencie modelu),

Layout Role rola występująca dla obiektów modelu będących kontenerami, określa między innymi jak zawarte w kontenerze obiekty mają być rozmieszczone na etapie wizualizacji,

Tree container role rola analogiczna do *Container role*, ale dotycząca widoku drzewiastego w edytorze,

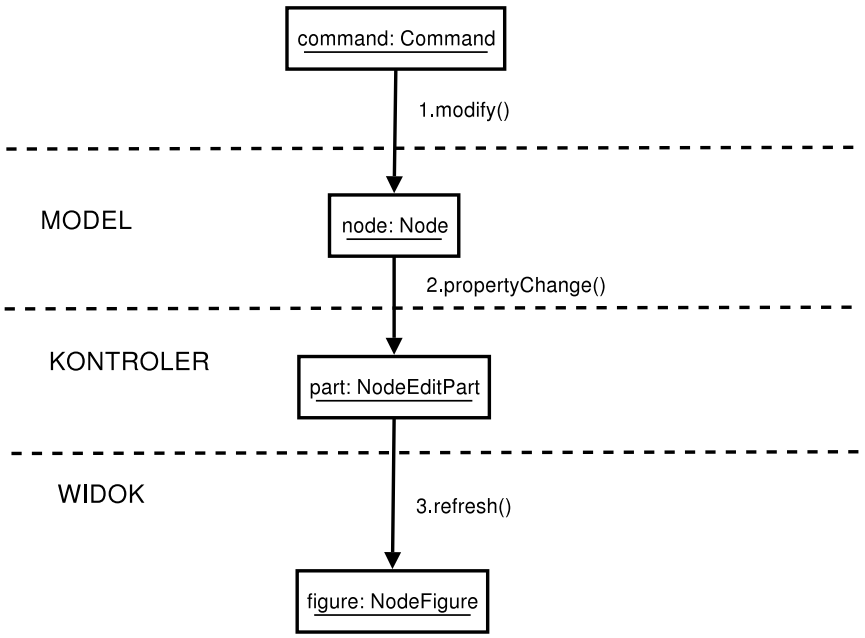
Graphical node role rola odpowiedzialna za tworzenie i zarządzanie połączeniami z perspektywy węzła (wymagana by elementy modelu można było łączyć),

Direct edit role rola pozwalająca użytkownikowi na bezpośrednią edycję właściwości elementu (np. zmiana nazwy elementu po dwukrotnym jej kliknięciu na diagramie),

Connection endpoints role rola pozwalająca użytkownikowi na zmianę obiektów, do których odnosi się dane połączenie.

Wiedząc już jak akcja użytkownika zostaje w środowisku *GEF* przetłumaczona na komendę, zwróćmy uwagę na interakcje zachodzące pomiędzy modelem, widokiem i kontrolerem (rys. 3.9):

1. Komenda modyfikuje obiekt modelu.
2. Kontroler zostaje poinformowany o zmianie modelu (wywołanie metody *propertyChange*).
3. Kontroler podejmuje modyfikację widoku.



Rysunek 3.9. Interakcje w *GEF*

3.4.3. Stworzenie prostego edytora

Zbudowany przez autorów edytor deskryptora *CAD* powstał w oparciu o dołączany do środowiska *GEF* przykładowy edytor kształtów (*shapes editor*), pozwalający łączyć ze sobą w graf skierowany dwa rodzaje węzłów: elipsy i prostokąty. Dysponując wiedzą z poprzedniego podpunktu zapoznamy się z kolejnymi krokami jakie należy wykonać by zbudować taki edytor w środowisku *GEF*.

W procesie budowy edytora wyróżniamy następujące etapy:

1. Stworzenie *wtyczki* do środowiska *Eclipse*.
2. Zdefiniowanie klasy głównej edytora.
3. Zdefiniowanie klas *modelu*.
4. Zdefiniowanie klas *widoku*.
5. Zdefiniowanie klas *kontrolera*.
6. Zdefiniowanie klas *polityk* i wykorzystywanych przez nich *komend*.

7. Stworzenie klasy fabryki elementów *kontrolera.*

Listing wszystkich klas omawianych w podpunkcie znajduje się w *Dodatku B.* Zamieszczony w dalszej części podpunktu opis powstał w oparciu o kod źródłowy oraz przewodnik do *Shapes Editor* (bibliografia: [17]).

Tworzenie wtyczki w Eclipse

Proces ten jest dobrze opisany (odsyłamy do bibliografii: [13]), dlatego poniżej przedstawiono jedynie schemat oraz parę istotnych uwag.

1. Tworzymy nowy projekt wtyczki w środowisku *Eclipse.*
2. Uzupełniamy listę wymaganych (*Dependencies)* paczek *jar* tak by zawierała:
   - org.eclipse.ui,
   - org.eclipse.ui.views,
   - org.eclipse.ui.ide
   - org.eclipse.core.runtime,
   - org.eclipse.core.resources,
   - org.eclipse.gef.
3. Importujemy kod zródłowy (dołączona do pracy płyta CD zawiera kod źródłowy *Shapes Editor* jak i gotowy do importu skonfigurowany projekt w środowisku *Eclipse*).
4. Modyfikujemy plik *plugin.xml* jak poniżej:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
     <extension point="org.eclipse.ui.editors">
      <editor name="Shapes Editor"
              extensions="shapes"
              icon="icons/sample.gif"
              default="true"
              class="org.eclipse.gef.examples.shapes.ShapesEditor"
              contributorClass="org.eclipse.gef.examples.shapes.
              ShapesEditorActionBarContributor"
              id="GEF Shapes Editor">
      </editor>
   </extension>
   <extension point="org.eclipse.ui.newWizards">
      <category name="Examples"
                parentCategory="org.eclipse.ui.Examples"
                id="org.eclipse.gef.examples"/>
      <wizard name="My Shapes Diagram"
              icon="icons/sample.gif"
              category="org.eclipse.ui.Examples/org.eclipse.gef.examples"
              class="org.eclipse.gef.examples.shapes.ShapesCreationWizard"
              id="org.eclipse.gef.examples.shapes.ShapesCreationWizard">
         <selection class="org.eclipse.core.resources.IResource"/>
      </wizard>
```

```
   </extension>
</plugin>
```

5. Modyfikujemy plik manifestu jak poniżej (istotne są *Bundle Activator* oraz *Require-Bundle*):

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: ShapesEditor Plug-in
Bundle-SymbolicName: shapesEditor; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: org.eclipse.gef.examples.shapes.ShapesPlugin
Bundle-Localization: plugin
Require-Bundle: org.eclipse.gef,
 org.eclipse.ui,
 org.eclipse.core.runtime,
 org.eclipse.core.resources,
 org.eclipse.ui.views,
 org.eclipse.ui.ide
Eclipse-AutoStart: true
```

Klasa główna edytora

ShapeEditor stanowi klasę główną edytora, dziedziczy po *GraphicalEditorWithFlyoutPalette* (jest to rodzaj graficznego edytora wyposażony w paletę narzędzi), co wymusza implementację dwóch metod abstrakcyjnych:

- *getPaletteRoot* - zwraca paletę edytora.
- *getPalettePreferences* - zwraca właściwości palety, tj. czy standardowo paleta jest widoczna i czy jest rozwinięta.

W konstruktorze edytora ustawiana jest domyślna (instancja *DefaultEditDomain*) implementacja *EditDomain,* która stanowi centrum kontrolera edytora, odpowiedzialne za zarządzanie paletą, narzędziami, stosem poleceń, obsługę zdarzeń generowanych przez użytkownika.

Głównym zadaniem *ShapeEditor* jest konfiguracja graficznej przeglądarki (*GraphicalViewer*) modelu co ma miejsce w metodzie *configureGraphicalViewer:*

- ustawiana jest fabryka (*ShapesEditPartFactory*) elementów kontrolera (szczegółowo opisana w dalszej części podpunktu),
- ustawiany jest jako *RootEditPart* obiekt klasy *ScalableFreeformRootEditPart* co zapewnia możliwość:
  — rozrastania się edytowanego diagramu we wszystkich kierunkach gdy użytkownik przestawi element poza dotychczasowe krawędzie diagramu,
  — przewijania,
  — skalowania.
- włączana jest standardowa (obiekt klasy *GraphicalViewerKeyHandler)* obsługa skrótów klawiszowych (np. klawisz "->","<-", "shift", "ctrl" itp),
- rejestrowane jest menu kontekstowe.

Skonfigurowana graficzna przeglądarka otrzymuje do wizualizacji model, co ma miejsce w metodzie *initializeGraphicalViewer*, edytor musi być w stanie wczytać poprawnie model jak i też zapisać modyfikacje - odbywa się to odpowiednio w metodach *setInput* oraz *doSave oraz doSaveAs.*

Przy omawianiu *EditDomain* został wspomniany stos poleceń, w metodzie *commandStackChanged* zmiana stosu wiąże się z wygenerowaniem zdarzenia informującego o potrzebie zapisu modelu, gdyż uległ on modyfikacji.

Kolejną istotną metodą jest *createTransferDropTargetListener* wzbogacająca edytor o możliwość przenoszenia elementów wprost z palety na diagram (mechanizm *drag&drop*).

Edytor poprzez implementację metody *getAdapter* wspiera wizualizacje modelu w postaci drzewa i korzysta w tym celu z klasy *ShapesOutlinePage* - szczegóły zostaną podane pod koniec podpunktu.

Klasy modelu

Jak wspomniano w podpunkcie 3.3.1 wszystkie elementy modelu muszą spełniać pewne wymagania, dlatego dobrze jest wspólną implementacje wydzielić do jednej klasy podstawowej (w tym przypadku jest to *ModelElement)*:

- metody *addPropertyChangeListener* oraz *removePropertyChangeListener* umożliwiają (za/od)rejerstrowanie obiektów nasłuchujących zmian stanu modelu,
- metoda *firePropertyChange* wywoływana jest by poinformować obiekty nasłuchujące o zmianie stanu modelu,
- implementacja interfejsu *java.io.Serializable* pozwala na odczyt/zapis binarny modelu,
- implementacja interfejsu *org.eclipse.ui.views.properties.IPropertySource* pozwala na integracje z widokiem *Properties* dostarczanym w platformie *Eclipse* (pełna implementacja zawarta w klasie *Shape*, dziedziczącej po *ModelElement*).

Jak wspomniano w edytorze występują dwa rodzaje kształtów: elipsy i prostokąty, wspólna dla nich implementacja została zawarta w klasie *Shape,* jest to:

- obsługa połączeń (zarówno wychodzących jak i kończących się w kształcie) - atrybuty *sourceConnections, targetConnections* oraz metody *addConnection*q, *removeConnection, getSourceConnections, getTargetConnections,*
- zmiana lokalizacji i rozmiaru kształtu - metody *set/getSize, set/getLocation* (warto zwrócić uwagę, że metody ustawiające te atrybuty, tzw. *setters* informują obserwatorów o zmianie poprzez wywołanie metody *firePropertyChange*),
- pełna implementacja interfejsu *org.eclipse.ui.views.properties.IPropertySource* (pozwalającego na edytowanie właściwości modelu poprzez *Properties View*) - metody *getPropertyDescriptors, getPropertyValue, setPropertyValue.*

Klasa *Connection* modeluje połączenie pomiędzy dwoma kształtami. Połączenie takie posiada następujące cechy:

- atrybuty *source, target* - pozwalają na tworzenie skierowanego połączenia,
- atrybut *lineStyle* - różnicuje sposób wizualizowania połączenia (linia ciągła, przerywana),

- atrybut *isConnected* - flaga oznaczająca czy połączenie jest prawidłowo utworzone pomiędzy dwoma kształtami,
- metody *reconnect, disconnect* - służą do utworzenia połączenia delegując to zadanie do wspomnianych metod *addConnection, removeConnection* w klasie *Shape,*
- podobnie jak klasa Shape, klasa Connection zawiera implementację interfejsu *org.eclipse.ui.views.properties.IPropertySource,* dzięki czemu właściwości połączenia można edytować w widoku *Properties.*

Klasy *RectangleShape* oraz *EllipseShape* zawierają elementy specyficzne dla poszczególnych kształtów (w tym przypadku są to: ikona kształtu oraz standardowy napis - metoda *toString*).

W końcowej fazie graf modelowany jest przez klasę *ShapesDiagram* będącą kontenerem dla kształtów, istotne są metody *addChild* i *removeChild,* które informują kontroler o potrzebie uaktualnienia widoku z uwzględnieniem nowo dodanego (usuniętego) elementu.

Klasy widoku

Widok do wizualizacji modelu korzysta ze standardowych klas dostarczancyh wraz z pakietem *Draw2D:*

- *org.eclipse.draw2d.Ellipse* (dla elipsy),
- org.eclipse.draw2d.*RectangleFigure* (dla prostokąta),
- *org.eclipse.draw2d.Freeformlayer (*dla diagramu - standardowa figura posiadająca właściwość nieograniczonego rozrastania się w czterech kierunkach).

Klasy kontrolera

Każda z klas kontrolera (*DiagramEditPart, ShapeEditPart, ConnectionEditPart*) implementuje w podobny sposób mechanizm notyfikacji: w metodach *activate oraz deactivate* ma miejsce odpowiednio zarejestrowanie/wyrejestrowanie się na powiadamianie o modyfikacji elementu modelu, w metodzie *propertyChange* podejmowana jest akcja mająca na celu uaktualnienie widoku w konsekwencji zmiany stanu modelu. W tabeli 3.1 przedstawiono szczegółowy opis istotnych szczegółów implementacyjnych kolejnych klas kontrolera.

Klasa kontrolera	Metoda	Uwagi
DiagramEditPart	*getModelChildren*	zwraca listę kształtów
	createFigure	tworzy widok diagramu za pomocą *Freeformlayer* oraz deleguje routing połączeń do instancji *ShortestPathConnectionRouter*
	propertyChange	uaktualnia widok całego diagramu w odpowiedzi na usunięcie / dodanie nowego kształtu
	createEditPolicies	instaluje polityki[2] dla elementu kontrolera: • *RootComponentEditPolicy* - uniemożliwia usunięcie elementu (diagram stanowi korzeń modelu), • *ShapesXYLayoutEditPolicy* - odpowiada za tworzenie, rozmieszczenie, zmiane rozmiaru i położenia kształtów na diagramie.
ShapeEditPart	*getSourceConnectionAnchor* *getTargetConnectionAnchor*	implementacja interfejsu *NodeEditPart*, metody określają w jaki sposób mają być mocowane połączenia do kształtów - w omawianym edytorze punktem mocowania jest punkt na obwodzie kształtu (standardowa implementacja *EllipseAnchor* oraz *ChopboxAnchor*)
	getModelSourceConnections *getModelTargetConnections*	metody te inforumują, jakie połączenia wychodzą a jakie się kończą w kontrolowanym elemencie modelu
	propertyChange	metoda uaktualnia widok w odpowiedzi na zmianę: rozmiaru, położenia, połączeń wchodzących i wychodzących

Tablica 3.1. Opis implementacji klas kontrolera

[2] implementacja polityk omówinona została w dalszej częsci podpunktu

Klasa kontrolera	Metoda	Uwagi
ShapeEditPart	*createFigure*	w zależności od rodzaju kształtu zwraca jako widok elipse lub prostokąt
	refreshVisuals	uaktualnia prezentowanie elementu w widoku przekazując jego aktualne położenie i rozmiar do kontrolera stowarzyszonego z danym elementem (czyli do *DiagramEditPart*), dbającego o prawidłowe rozmieszczenie kształtów
	createEditPolicies	instaluje polityki dla elementu kontrolera: • *ShapeComponentEditPolicy* - zapewniającą możliwość usunięcia kształtu, • *GraphicalNodeEditPolicy* - odpowiada za łączenie i rozłączanie kształtów.
ConnectionEditPart	*createFigure*	do wizualizacji połączenia stosuje standardową klasę *PolylineConnection*
	propertyChange	metoda uaktualnia widok w odpowiedzi na zmianę stylu połączenia (linia ciągła albo przerywana)
	createEditPolicies	instaluje polityki dla elementu kontrolera: • *ConnectionEditPolicy* - zapewniającą możliwość usunięcia połączenia, • *ConnectionEndpointEditPolicy* - wzbogaca element o uchwyty, pozwalające na zmianę podłączonego kształtu.

Tablica 3.2. Opis implementacji klas kontrolera (ciag dalszy)

Klasa fabryki elementów kontrolera

ShapesEditPartsFactory implementuje metodę *createEditPart* zwracającą odpowiedni element kontrolera w zależności od obiektu modelu, dla którego *EditPart* jest żądany. Metoda ta jest wywoływana przez silnik (*engine)* środowiska *GEF* dla każdego elementu zawartego na liście zwracanej przez metodę getModelChildren omawianej klasy *DiagramEditPart,* a następnie stworzony *EditPart* tworzy widok kontrolowanego obiektu z modelu.

Klasa polityki	Klasa komendy	Działanie
ShapeComponentEditPolicy	*ShapeDeleteCommand*	usunięcie kształtu
ConnectionEditPolicy	*ConnectionDeleteCommand*	usunięcie połączenia
GraphicalNodeEditPolicy	*ConnectionCreateCommand*	stworzenie połączenia
	ConnectionReconnectCommand	skojarzenie połączenia z nowym źródłem lub końcem
ShapesXYLayoutEditPolicy	*ShapeCreateCommand*	stworzenie kształtu
	ShapeSetConstraintCommand	zmiana położenia lub rozmiaru kształtu

Tablica 3.3. Polityki i komendy

Klasy polityk oraz komend

Jak już powiedziano polityki wykorzystywane są do zamiany żądań na komendy, tabela 3.3 przedstawia zestawienie komend generowanych przez poszczególne polityki.

Tworzenie komend wymaga zaimplementowania następujących metod:

execute zawiera funkcjonalność komendy,
canExecute określa w jakich warunkach komenda może zostać wykonana,
canUndo określa w jakich warunkach komenda może zostać cofnięta,
undo określa w jaki sposób następuje cofnięcie komendy,
redo określa w jaki sposób komenda zostaje ponownie wykonana.

Wizualizacja modelu za pomocą struktury drzewiastej

Na koniec jako pewna dodatkowa funkcjonalność zostanie omówiona wizualizacja modelu w postaci drzewa. Jak wcześniej powiedziano metoda *getAdapter* z klasy głównej edytora *ShapesEditor* by stworzyć widok drzewiasty korzysta *z ShapesOutlinePage*.

W metodzie *createControl* klasy *ShapesOutlinePage* odbywa się analogiczny proces do przedstawionego dla klasy głównej edytora, tj:

- ustawiany jest ten sam co dla *ShapesEditor* obiekt klasy *EditDomain*,
- ustawiana jest fabryka elementów kontrolera *ShapesTreeEditPartsFactory.*

Wspomniana fabryka *ShapesTreeEditPartsFactory* pełni analogiczną rolę jak już wcześniej omówiona *ShapesEditPartsFactory* z tą różnicą, że zwraca w metodzie *createEditPart* "drzewiaste" *elementy* kontrolera, tzn. instancje klas *ShapeTreeEditPart oraz DiagramTreeEditPart.*

Jak widać niewielkim kosztem można uzyskać zupełnie nowe spojrzenie (widok) na edytowany model.

3.5. GEF w środowisku Eclipse

Eclipse jest rozbudowanym środowiskiem programistycznym napisanym w *Javie*. Projekt został stworzony przez firmę *IBM*, a następnie udostępniony na zasadach otwartego oprogramowania. W chwili obecnej jest on rozwijany przez *Fundację Eclipse*.

Sama platforma nie dostarcza żadnych narzędzi służących do tworzenia kodu i budowania aplikacji, oferuje jednak obsługę wtyczek (ang. *plugin*) rozszerzających jej funkcjonalność, umożliwiających m.in. rozwijanie aplikacji w językach *Java*, *C/C++*, *PHP*, tworzenie *GUI*, modelowanie aplikacji za pomocą *UML*, współpracę z serwerami aplikacji, serwerami baz danych, itp. Wszystkie wtyczki wchodzące w skład środowiska *Eclipse* dostarczane są w postaci pluginów, automatycznie rozpoznawanych i dołączanych do środowiska. Tworzeniem pluginów zajmuje się wiele firm, organizacji i osób prywatnych, co zapewnia stały dopływ nowych modułów rozszerzających *Eclipse* o nowe możliwości. Tworzenie rozszerzeń jest możliwe za sprawą *Eclipse Plug-in Development Environment (PDE)*. Aby zainstalować nową wtyczkę należy skopiować rozpakowane archiwum aplikacji do katalogu *$ECLIPSE_HOME*[3]*/plugins*. Po restarcie *Eclipse* powinien umożliwiać korzystanie z zainstalowanych *wtyczek*. W zależności od rozszerzenia, dostępne bedą nowe perspektywy, opcje w menu lub inne możliwości.

Istnieje możliwość tworzenia własnych rozszerzeń tego środowiska. Każda wtyczka posiada deskryptor w postaci pliku *.xml* (plik *plugin.xml*) pozwalający na określenie współdziałania budowanej wtyczki ze środowiskiem, w którym ma działać oraz określenie klas implementujących zadaną funkcjonalność. Dane w pliku deskryptora zebrane są w *punktach rozszerzeń* (ang. *extension point*). Każdy z tych punktów posiada swoją unikatową nazwę oraz typ określający pełnioną funkcję.

Tworzenie wtyczki do *Eclipse* sprowadza się do wykorzystania kontrolek z pakietu *SWT (Standard Widget Toolkit)* wykorzystywanych przez środowisko *Eclipse*. W przypadku tworzenia edytora tekstowego istnieje możliwość wykorzystania klasy abstrakcyjnej *org.eclipse.ui.texteditor.AbstractTextEditor* i dodefiniowania pożądanej funkcjonalności. Stworzenie tej klasy abstrakcyjnej było możliwe ze względu na fakt, że wszystkie edytory tekstowe posiadają pewną standardową funkcjonalność, tak więc w tym przypadku autor wtyczki musi określić tylko niestandardową funkcjonalność, jeśli taką przewiduje w swoim edytorze. W przypadku edytorów graficznych sprawa nie jest już taka prosta. Trudno jest określić podstawową funkcjonalność edytora tego typu. Znane są jednak ogólne cechy, jakie edytor graficzny może posiadać. Do takich wymagań funkcjonalnych można zaliczyć wykorzystanie myszki jako narzędzia do interakcji z użytkownikiem (np. przesuwania obiektów), mechanizm "przeciąg-upuść", wykorzystanie palety, itp. W celu usprawnienia tworzenia różnorakich edytorów graficznych stworzono *GEF'a,* który jest zbiorem bibliotek ułatwiających tworzenie edytorów graficznych.

Aplikacja, która powstała razem z tym dokumentem jest wtyczką środowiska *Eclipse* przy wykorzystaniu środowiska *GEF*. Dzięki temu stworzony projekt cechuje się bogatą funkcjonalnością związaną zarówno z problem generowania pliku *.cad* jak i obsługą interfejsu graficznego.

3.6. Podsumowanie

Pojawienie się środowiska *Eclipse* i jego rozszerzeń znacznie ułatwiło programowanie w języku *Java*. Jak się okazuje, dzięki bogatej puli dostępnych rozszerzeń możliwe jest wykorzystanie aplikacji *Eclipse* do różnych celów. Korzystając z środowiska *GEF* oraz

[3] katalog domowy programu *Eclipse*

udogodnień dostarczanych przez środowisko *Eclipse* stworzenie aplikacji do graficznego tworzenia deskryptora *.cad* stało się zadaniem znacznie łatwiejszym do wykonania. *GEF* jest bardzo złożony, co powoduje, że poznanie wszystkich tajników związanych z programowaniem wykorzystującym go jest czasochłonne, ale dzięki bogatej funkcjonalności zapewnianej przez środowisko *GEF* mogliśmy się skupić na dziedzinie problemu, a nie na rozwiązywaniu kwestii implementacyjnych związanych z zachowaniem edytora i obsługą podstawowych akcji wywoływanych przez użytkownika.

Rozdział 4

CGE - CAD Graphical Editor

CORBA Component Model jest bardzo funkcjonalnym, ale jednocześnie złożonym środowiskiem. Proces osadzania stworzonej aplikacji wymaga wielu deskryptorów. Najważniejszym, ale jednocześnie najbardziej rozbudowanym plikiem konfiguracyjnym jest plik o rozszerzeniu *cad (Component Assembly Descriptor).* Wskazane jest aby deskryptor ten był generowany przez graficzne narzędzie. W ramach tej pracy powstała aplikacja o nazwie *CGE* (*CAD Graphical Editor*), która pozwala na graficzne tworzenie deskryptora asemblacji *cad.* Edytor korzysta z biblioteki *GEF*, która została opisana w poprzednim rozdziale. Niniejszy rozdział opisuje funkcjonalność stworzonej aplikacji.

4.1. Analiza wymagań systemu *CGE*

Ze względu na nowatorski charakter tworzonego narzędzia, projektowanie jego funkcjonalności było procesem iteracyjnym. Prace implementacyjne sprowadzały się do tworzenia stale modyfikowanego prototypu, który miał na celu zbadanie jakie elementy aplikacja ma zawierać oraz wybór rozwiązania dla każdej rozwijanej funkcjonalności. Ze względu na złożoność oraz uniwersalność deskryptora *.cad,* z gamy znaczników opisanych w jego specyfikacji musieliśmy wybrać pewien podzbiór elementów do obsługi przez nasz edytor. Kolejna sekcja dokładnie opisuje końcowy zbiór postawionych wymagań wobec narzędzia.

4.1.1. Zbiór wymagań funkcjonalnych

Przed tworzoną aplikacją postawiono następujące wymagania:

1. **Graficzna reprezentacja komponentów i ich portów**
   Podstawowym zadaniem aplikacji *CGE* jest reprezentacja na diagramie elementów używanych w deskryptorze asemblacji. Konwencja graficznego przedstawienia

komponentów i ich portów powinna nawiązywać do tej zaproponowanej przez twórców modelu *CCM* (patrz [1]). Komponent zawiera pewne właściwości tekstowe, rozmiary, położenie na diagramie oraz zestaw związanych z nim portów. Najważniejsze właściwości tekstowe komponentu (tj. nazwa i typ) powinny być widoczne na reprezentującym go elemencie diagramu. Tworzenie komponentu powinno być intuicyjne i szybkie, a pola jego właściwości powinny być w miarę możliwości wypełniane automatycznie.

Reprezentacja portów związanych z danym komponentem powinna być czytelna; kształt i kolor portu reprezentują jego typ. Z danym portem stowarzyszone są jego właściwości tekstowe informujące m.in. o rodzaju portu i nazwie.

Tworzony diagram może zawierać wiele elementów. Dlatego bardzo ważne jest, aby edytor dawał możliwość przesuwania komponentów w obrębie diagramu oraz portów pomiędzy poszczególnymi krawędziami elementu, do którego należą. W obu przypadkach zaleca się wykorzystanie myszki do zmiany położenia elementu.

2. **Intuicyjna reprezentacja miejsc alokacji *(Destination)***

   Dla każdego komponentu istnieje potrzeba określenia jego środowiska wykonania jakim jest serwer komponentów lub węzeł (*node*). W celu uogólnienia nazwy środowiska wykonania komponentu wprowadźmy pojęcie *miejsca alokacji* (*destination*), przez które będziemy rozumieć serwer komponentów lub węzeł. Przypisanie *miejsca alokacji* do komponentu musi być odzwierciedlane w czytelny sposób na diagramie, jednocześnie zapewniając łatwą jego modyfikację. Wymagana jest możliwość grupowego przypisywania oraz domyślny wybór *miejsca alokacji* dla każdego komponentu. Edytor powinien zapewnić współistnienie na jednym diagramie wielu instancji tego samego kontenera, co ma na celu poprawę przejrzystości modelu. Wymóg ten wynika z faktu, że komponenty znajdujące się w różnych częściach diagramu mogą należeć do tego samego *miejsca alokacji*. Powinna istnieć możliwość takiego sposobu przypisywania komponentowi jego kontenera, aby wcześniej narysowany zestaw komponentów nie wymagał przeorganizowania podczas procesu przypisywania.

   Działające środowisko *CCM* charakteryzuje się dynamiką, co odzwierciedla się tym, że ilość działających miejsc alokacji jest zmienna w czasie. Ze względu na ten fakt, potrzebne jest rozwiązanie umożliwiające wstępne pogrupowanie komponentów. Grupy takie mogą być nazwane *logicznymi miejscami alokacji (Logical Destination* - w skrócie *LDS*). LDS umożliwia komponentom przynależność do pewnego, nie określonego w fazie projektowania aplikacji rozproszonej, wirtualnego serwera komponentów. Rozwiązanie takie pozwala na korzystanie z edytora bez wczytania informacji o środowisku (w języku angielskim tryb taki bywa określany jako *off-line*). W chwili gdy uzyskamy informacje o środowisku możliwe będzie przypisanie wyspecyfikowanym wcześniej grupom konkretnych, fizycznych miejsc alokacji. Dla odróżnienia od LDS dla określenia fizycznego miejsca alokacji będziemy używać pojęcia *PhDS* (*Physical Destination*).

3. **Wielopoziomowa prezentacja diagramu**

   Ze względu na wprowadzone pojęcia fizycznego i logicznego miejsca alokacji komponentów stawia się wymaganie prezentowania tworzonego diagramu na 3 sposoby:

   prosty - miejsca alokacji są całkowicie **niewidoczne** - widok ten ma za zadanie przedstawienie modelu w jak najprostszy sposób (z jak najmniejszą ilością

ozdobników) z uwzględnieniem wykorzystywanych komponentów i utworzonych połączeń między ich portami,

logiczny - wzbogaca *widok prosty* o **logiczne** miejsca alokacji - widok ten zawiera dodatkowo informacje o przypisaniu komponentów do logicznych miejsc alokacji; jest użyteczny szczególnie w sytuacji, gdy korzystamy z edytora bez wczytywania informacji o fizycznych miejscach alokacji,

fizyczny - wzbogaca *widok prosty* o **fizyczne** miejsca alokacji - widok ten zawiera dodatkowo informacje o przypisaniu komponentów do logicznych miejsc alokacji.

Każdy z widoków powinien zawierać charakterystyczne elementy wizualne, tak aby użytkownik bez problemu rozpoznawał w jakim trybie aktualnie pracuje.

4. **Różne sposoby prezentacji diagramu**
   Tworzony edytor powinien umożliwiać prezentowanie diagramów i operowanie na nich w przyjazny dla użytkownika sposób, nawet tych najbardziej złożonych. W przypadku gdy tworzony model nie mieści się na ekranie pojawia się możliwość przesunięcia ekranu za pomocą suwaków. Wskazane jest też wprowadzenie funkcji przybliżania/oddalania diagramu (*zoom in/out*). Istnienie alternatywnego sposobu przedstawienia modelu jest także pożądane. Prezentowanie struktury elementów umieszczonych na diagramie najlepiej odda postać drzewa elementów. Całościowy pogląd na diagram można by osiągnąć poprzez prezentowanie diagramu w postaci miniatury.
5. **Pobranie definicji komponentów i miejsc alokacji**
   Jedną z podstawowych funkcjonalności wymaganych od tworzonego edytora jest możliwość połączenia do zdalnego repozytorium i pobranie z niego definicji komponentów. Powinna także istnieć możliwość połączenia do serwisu nazw i pobranie z niego informacji o dostępnych miejscach alokacji. Proces połączenia powinien być intuicyjny oraz uniwersalny.
6. **Walidacja połączenia pomiędzy portami komponentów**
   Edytor powinien umożliwiać połączenia między sobą portów komponentów. Podczas procesu łączenia portów musi zachodzić walidacja połączenia, w celu sprawdzenia czy łączone porty są zgodnych typów. Gdy dane połączenie nie jest dozwolone użytkownik powinien zostać poinformowany o przyczynie niekompatybilności.
7. **Wczytanie wcześniej rozmieszczonych komponentów**
   Proces asemblacji aplikacji rozproszonej może być procesem iteracyjnym. Dlatego tworzony edytor powinien mieć możliwość wczytania dostępnych, aktualnie osadzonych komponentów w systemie. Ze względu na swój specjalny charakter wczytane komponenty winny się odróżniać wizualnie od innych.
8. **Odczyt i zapis diagramu z/do pliku**
   W edytorze wymagana jest funkcja odczytu i zapisu diagramu do pliku binarnego. Plik taki w odróżnieniu od pliku *.cad* zawiera nie tylko informacje semantyczne reprezentowane przez diagram, ale także dane dotyczące samego modelu i jego elementów np. położenie komponentów, portów, wybrany widok, itp. Rozwiązanie takie pozwala na zapis projektu w każdym momencie pracy, bez względu na to w jakim stadium znajduje się budowa diagramu i czy zbudowany diagram jest poprawny.
9. **Możliwość jednoczesnej pracy na wielu diagramach**

Wymagana jest możliwość jednoczesnego otwarcia i pracy na wielu diagramach. Rozwiązanie takie pozwala na porównywanie między sobą projektowanych aplikacji rozproszonych, czy też testowanie różnych konfiguracji. Każdy z otwartych diagramów cechuje się własnymi ustawieniami. Zmiany na jednym diagramie nie powinny mieć wpływu na inny otwarty diagram.

10. **Dodawanie elementów do diagramu za pomocą palety**
    W edytorze zalecana jest obecność palety - miejsca gdzie zgromadzone byłyby wszystkie narzędzia pomocne w tworzeniu i modyfikacji diagramu. Paleta zawierać miałaby także obiekty reprezentujące dane wczytane ze środowiska - fizyczne czy też logiczne miejsca alokacji oraz komponenty (także te aktualnie osadzone). Korzystanie z palety sprowadza się do czerpania korzyści z metody przeciąg-upuść (*drag and drop*).
11. **Obsługa stosu poleceń (undo & redo)**
    Aby zapewnić przyjazną użytkownikowi obsługę diagramu zaleca się wykorzystanie stosu poleceń umożliwiającego cofanie i ponawianie ostatnich akcji w edytorze (znane jako *undo-redo).* Obsługa stosu jest popularna prawie w każdym systemie operacyjnym oraz programie wykorzystującym graficzny interfejs użytkownika. Dlatego ważne jest, aby obsługa ta odbywała się w sposób zgodny z przyjętymi standardami.
12. **Zróżnicowane tryby zaznaczania elementów**
    Diagram zawiera wiele typów elementów. Aby przyspieszyć dokonywanie zmian na wielu obiektach jednocześnie powinna istnieć możliwość zaznaczania wielu obiektów i dokonywania zmian na zaznaczonym zbiorze. Lecz ze względu na fakt, że różne typy obiektów charakteryzują się różnymi własnościami, powinna istnieć możliwość zaznaczania tylko wybranych typów elementów z pominięciem innych.
13. **Drukowanie diagramu**
    Czasem konieczne jest zaprezentowanie projektowanej aplikacji rozproszonej bez użycia komputera. Dlatego też zalecane jest stworzenie możliwości drukowania diagramu.
14. **Generowanie pliku *.CAD***
    Tworzony za pomocą edytora diagram należy zapisać jako plik *XML* o rozszerzeniu *.CAD.* Generowany plik musi być zgodny z przyjętym przez grupę *OMG* standardem, tak aby mógł być wykorzystywany przez inne aplikacje środowiska *CCM.* Edytor powinien mieć możliwość określenia katalogu na dysku twardym, gdzie wygenerowany plik ma być zapisany. Zalecana jest także możliwość wskazania pliku archiwum *.AAR,* do którego tworzony plik ma zostać automatycznie wgrany.

4.1.2. Zbiór wymagań niefunkcjonalnych

Tworzony edytor powinien spełniać następujące wymagania niefunkcjonalne:

1. **Przenośność**
   Edytor powinien pracować zarówno na systemach Windows, Unix, jak i Solaris.
2. **Wykorzystanie istniejących komponentów**
   W celu przyspieszenia implementacji zalecane jest wykorzystanie w budowie edytora w jak najwięcej istniejących bibliotek i środowisk umożliwiających realizację

wymagań funkcjonalnych. Wskazane jest by korzystać jedynie z darmowego oprogramowania.

3. **Optymalizacja pamięciowa**
   Edytor powinien mieć możliwie najniższe wymogi pamięciowe, tak by do korzystania z narzędzia wystarczył średniej klasy komputer PC.

4.1.3. Przyjęte ograniczenia

Z powodu istniejących wymogów czasowych wobec implementacji edytora przyjęto następujące ograniczenia:

1. Edytor współpracować będzie z jedną implementacją modelu *CCM - OpenCCM.*
2. Źródło wiedzy o miejscach alokacji stanowi wyłącznie *Naming Service*.
3. Brak możliwości definiowania komponentów przez użytkownika (ograniczenie źródła definicji do *Interface Repository*).
4. Implementacja podzbioru schematu deskryptora *CAD* (opisanego w *Dodatku A).*
5. Brak zapamiętywania (brak *cache w* edytorze) informacji semantycznej wymaganej do walidacji portów (nie przewiduje się możliwości łączenia portów w trybie pracy *offline*).
6. Brak importu istniejącego deskryptora *CAD* do edytora.

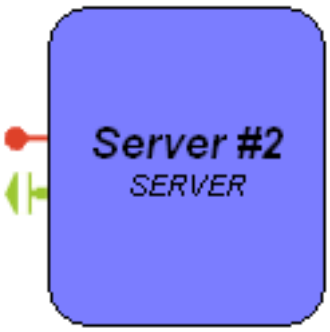
4.2. Realizacja przyjętych wymagań - projekt i implementacja

Dzięki ogromnym możliwościom środowiska *GEF* zrealizowaliśmy opisywaną funkcjonalność edytora. Pozostała część niniejszego rozdziału stanowi dokładny opis zastosowanych rozwiązań pozwalających na wypełnienie stawianych wymagań. Poszczególne sekcje przedstawiają wyodrębniony w wymaganiu fragment funkcjonalności oraz sposób jego implementacji.

4.2.1. Graficzna reprezentacja komponentów *CCM* i ich portów

W edytorze *CGE* komponenty reprezentowane są jako niebieskie zaokrąglone prostokąty. Obiekty te tworzy się przeciągając wczytane wcześniej z repozytorium interfejsów obiekty znajdujące się na palecie (*Palette*). Z każdym komponentem stowarzyszona jest jego nazwa oraz nazwa jego typu. Rysunek 4.1 przedstawia przykładowy komponent z dwoma portami. Nazwa obiektu to *Server #2,* a jego typ to *SERVER.*

Pozostałe własności obiektu zawiera okno właściwości (*Properties View*). Każdy nowo tworzony komponent posiada właściwości wypełnione tymi samymi wartościami co poprzedni egzemplarz tego typu obiektu. W przypadku, gdy dany komponent jest tworzony po raz pierwszy automatycznie wypełniane pola zawierają standardowe wartości. Większość właściwości dotyczy tylko wybranego komponentu. Przykładem właściwości wspólnej dla wszystkich obiektów jest: *Component Assembly ID.* Zmiana tej wartości dla jednego komponentu zmienia ją dla wszystkich obiektów diagramu.



Rysunek 4.1. Przykładowy komponent

Istnieje możliwość zmiany wybranej właściwości dla większej ilości obiektów jednocześnie. W tym celu należy zaznaczyć za pomocą myszki wybrane obiekty; okno właściwości *Properties* pokaże tylko właściwości wspólne wybranych obiektów.

Porty należące do danego komponentu znajdują się na jego krawędziach. Aby przesunąć wybrany port na inną krawędź komponentu można przeciągnąć go za pomocą myszki lub klikając na niego wybrać położenie z okna właściwości. Możliwe są 4 położenia portu:

- *WEST* - zachód,
- *SOUTH* - południe,
- *EAST* - wschód,
- *NORTH* - północ.

Właściwości tekstowe portu pokazują się jako wskazówka (*tool-tip*) po najechaniu kursorem myszki na wybrany port. Rysunek 4.2 przedstawia wyświetlane wartości (nazwa, nazwa absolutna[1], nazwa typu, absolutna nazwa typu oraz rodzaj portu).

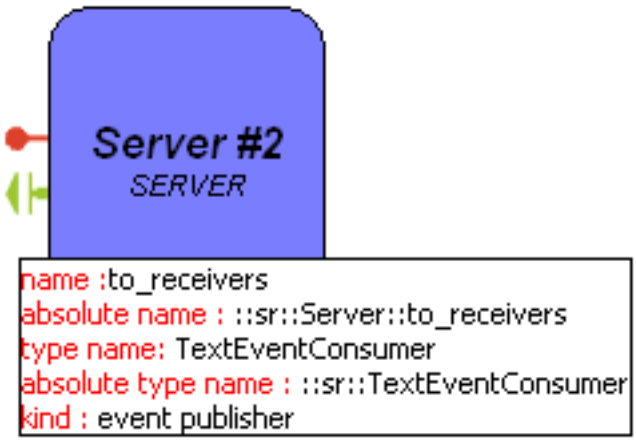
Edytor obsługuje 5 rodzajów portów:

- *facet,*
- *receptacle,*
- *event emitter,*
- *event publisher,*
- *event sink.*

Rysunek 4.3 przedstawia ikonki każdego rodzaju portu (przedstawione zgodnie z kolejnością z powyższej listy zaczynając od lewej).

Rysunek 4.3. Ikonki portów

[1] nazwa lub typ absolutny składa się z pełnej listy pakietów, w której dany typ / nazwa zostały zdefiniowane



Rysunek 4.2. Przykładowe właściwości portu

Prace związane z realizacją wymagania dotyczącego graficznej reprezentacji komponentów i ich portów zostało zrealizowane poprzez rozbudowę i modyfikację opisywanego w poprzednim rodziale edytora *Shapes Editor*.

Modyfikacje modelu

Rysunek 4.4 przedstawia strukturę modelu; graficzne zamodelowanie deskryptora *CAD* wiązało się z licznymi modyfikacjami edytora *Shapes Editor,* co wymagało implementacji szeregu nowych klas. Oto krótkie omówienie funkcjonalności nowych elementów:

CCMDiagram - klasa modelująca diagram, agreguje komponenty oraz połączenia między ich portami, przechowuje także, m.in:
- wybrany widok,
- parametry połaczeń z serwisami *CCM*,
- referencję do repozytorium miejsc alokacji.

Elementy te zostaną wkrótce dokładniej omówione w opisie realizacji pozostałych wymagań.

Component - klasa zawierająca wspólne dla dwóch rodzajów komponentów (*Deployed* i *CAD Component)* cechy, takie jak nazwa oraz zestaw posiadanych portów.

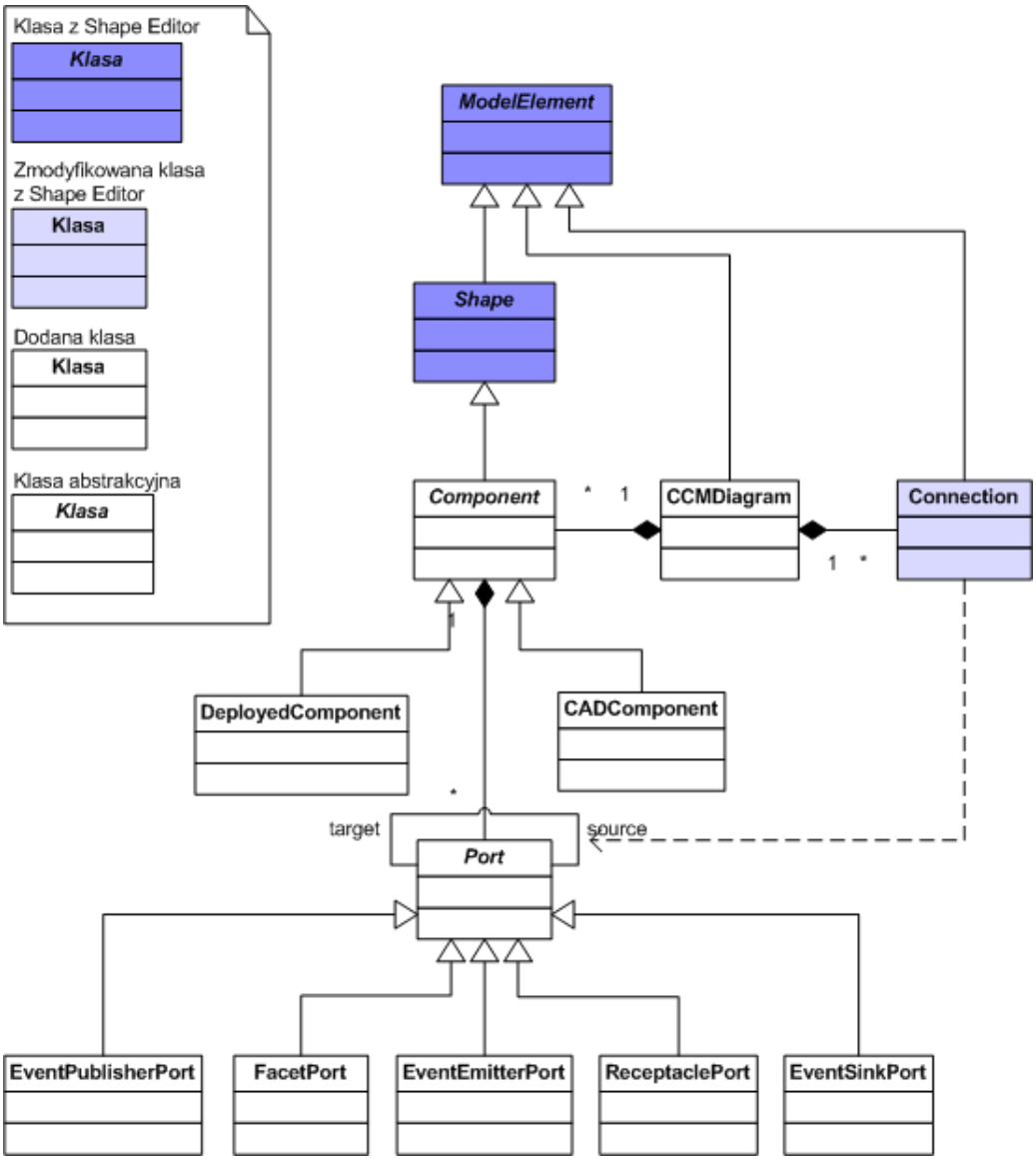
CADComponent - klasa modelująca komponent, który zostanie dopiero utworzony w wyniku uruchomienia aplikacji zamodelowanej w edytorze. W stanie instancji tej klasy zawarta jest cała informacja niezbędna do wygenerowania pliku *CAD* (agreguje zarówno wartości opisujące komponent jak i jego *home*).

DeployedComponent - klasa reprezentująca komponent, który został utworzony i osadzony w trakcie lub przed rozpoczęciem tworzenia diagramu. Modelowane komponenty charakteryzuje dynamiczny cykl życia (mogą się pojawiać i znikać w trakcie pracy w edytorze), dlatego też narzędzie musi je specjalnie traktować, o czym mowa jest w sekcji 4.2.6 na 70 stronie.

Port - klasa modelująca cechy wspólne portu komponentu; jej stan opisany jest poprzez nazwę oraz typ danego portu, a także komponent, w skład którego wchodzi.

FacetPort - klasa reprezentująca port typu *facet.*

ReceptaclePort - klasa reprezentująca port typu *receptacle*. Wspiera dwa typy tego portu zarówno *single* (podłączany do jednego portu typu *facet*) jak i *multiple* (podłączany do wielu portów typu *facet*).



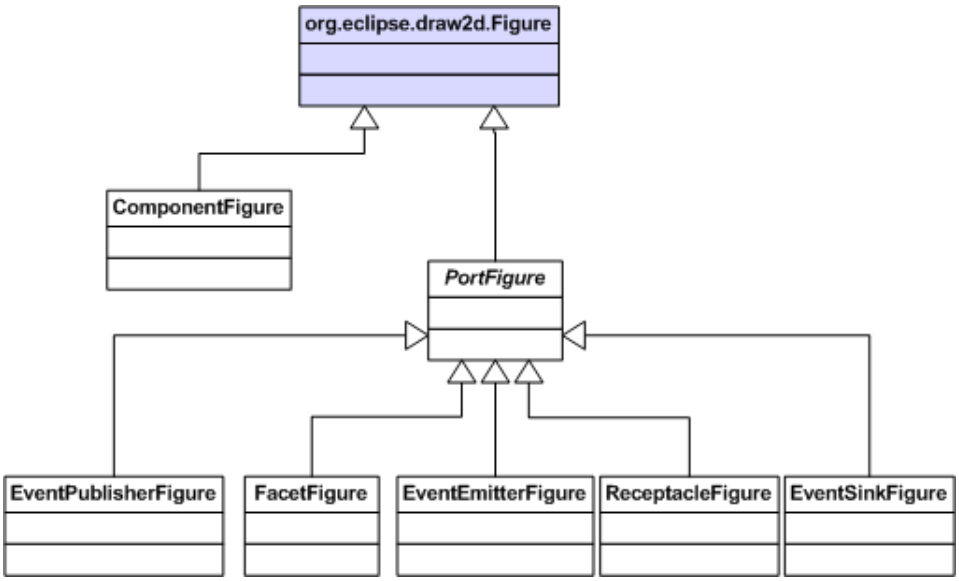
Rysunek 4.4. Diagram klas implementacji wzorca *Model - Widok - Kontroler*: klasy modelu (uwzględnione zostały klasy wykorzystywane do realizacji omawianego wymagania)

EventEmitterPort - klasa reprezentująca port *event emitter*.
EventPublisherPort - klasa reprezentująca port *event publisher*.
EventSinkPort - klasa reprezentująca port *event sink*.

Modyfikacja widoku

W edytorze *Shape Editor* widok w zasadzie nie zawierał żadnych specyficznych klas (wykorzystywał zawarte kształty w środowisku *GEF*). Dla potrzeb widoku edytora

CGE wymagane było stworzenie klas przedstawionych na rysunku4.5 (nazwy klas odpowiadają elementom modelu, dla którego stanowią widok).



Rysunek 4.5. Diagram klas implementacji wzorca *Model - Widok - Kontroler*: klasy widoku

Modyfikacja kontrolera

Rysunek 4.6 przedstawia klasy rozbudowanego dla edytora *CGE* kontrolera. Widoczne są dwie hierarchie w obrębie kontrolera:

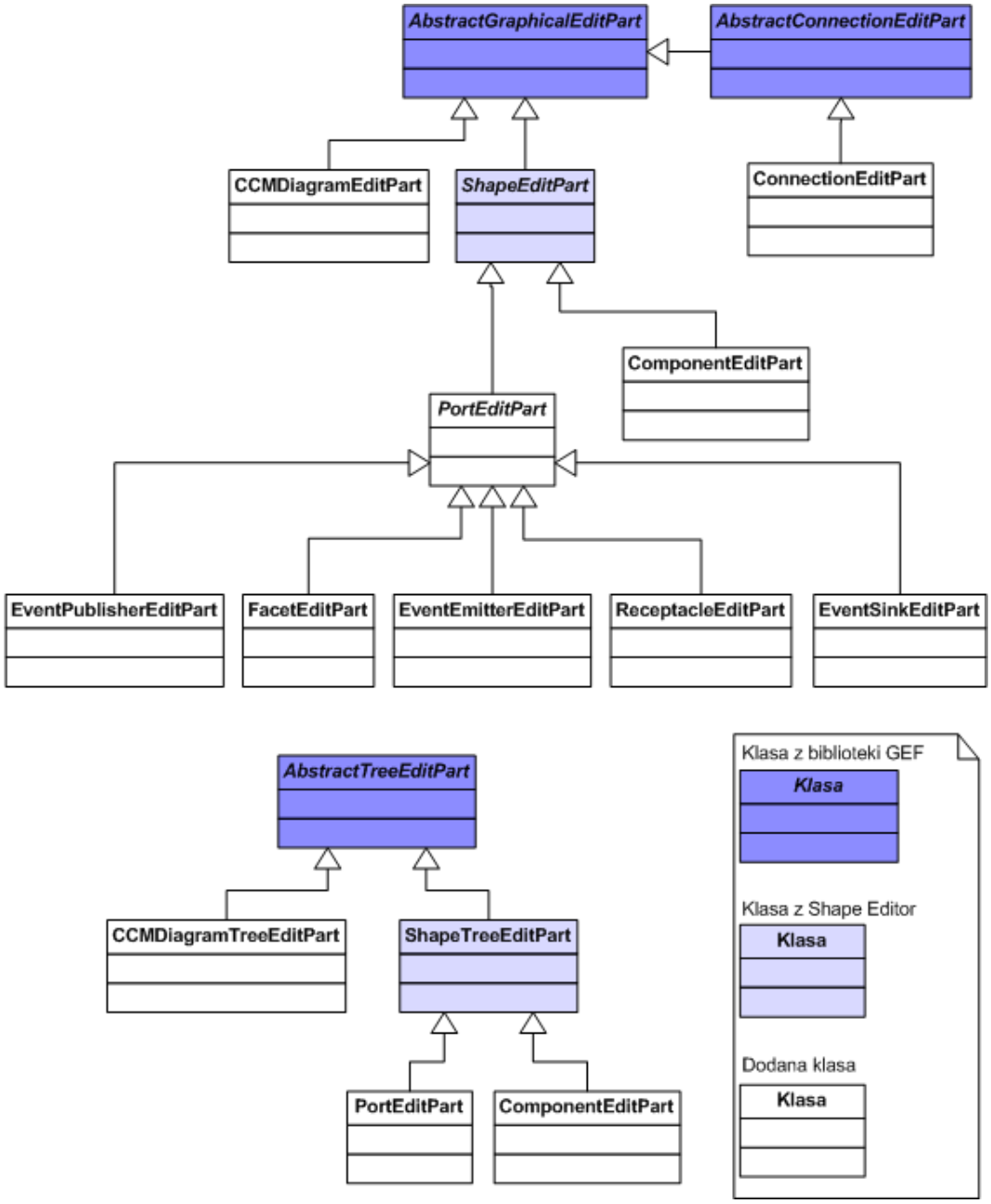
- część kontrolująca model wizualizaowany w postaci diagramu (korzeń stanowi klasa *AbstractGraphicalEditPart),*
- część kontrolująca model wizualizowany w postaci drzewa (korzeń stanowi klasa *AbstractTreeEditPart*).

Warto zatrzymać się w tym miejscu nad pierwszą zdecydowanie bardziej rozbudowaną częścią. Najbardziej istotne są trzy klasy:

CCMDiagramEditPart - element kontrolera odpowiedzialny za obsługę zdarzeń dodania nowego, usunięcia istniejącego komponentu z diagramu oraz zmiany aktualnego widoku diagramu (widoki zostaną omówione w 4.2.3 na 61 stronie),

ComponentEditPart - element kontrolera odpowiedzialny za aktualizacje widoku komponentu,

PortEditPart - element kontrolujący port, obsługujący zdarzenie zmiany lokalizacji portu w obrębie danego komponentu (klasy *EventPublisherEditPart, EventEmitterEditPart, EventSinkEditPart, FacetEditPart, ReceptacleEditPart* dziedzicząc po *PortEditPart* przesłaniają jedynie metody odpowiedzialne za wizualizacje modelu).



Rysunek 4.6. Diagram klas implementacji wzorca *Model - Widok - Kontroler*: klasy kontrolera

Modyfikacja fabryk elementów kontrolera

Zmiany sprowadziły się do dodania nowych klas kontrolera, zwracanych dla odpowiednich elementów modelu (np. dla *FacetPort - FacetEditPart).*

Modyfikacja polityk i komend kontrolera

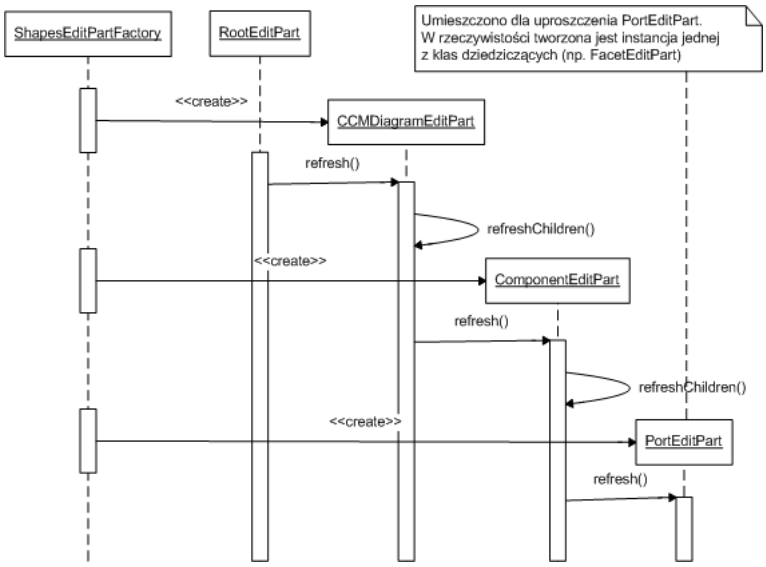
Polityki nie uległy większym zmianom zwiększyła się natomiast liczba komend - aktualna lista wygląda następująco:

- *znane komendy:*

  - *AddCommand* (dodanie komponentu),
  - *ComponentDeleteCommand* (usunięcie komponentu),
  - *ConnectionCreateCommand* (stworzenie połączenia),
  - *ConnectionDeleteCommand* (usunięcie połączenia),
  - *ConnectionReconnectCommand* (przełączenie istniejącego połączenia),
  - *SetConstraintCommand* (zmiana położenia lub rozmiaru komponentu albo portu),
  - *ShapeCreateCommand* (utworzenie kształtu).
- *nowe komendy* (dokładny opis nowych poleceń zamieszczony jest wraz z wymaganiami, do realizacji których zostały wykorzystane):
  - *AssignDestinationCommand,*
  - *ChangeCCMServicesSettingsCommand,*
  - *ChangeViewCommand,*
  - *EditLogicalDestCommand,*

Wizualizacja modelu

Po zapoznaniu się z modyfikacjami wprowadzonymi do *Shape Editor* warto zobaczyć jak poszczególne elementy współpracują ze sobą w celu wizualizacji modelu. Rysunek 4.7 przedstawia diagram sekwencji obrazujący ten proces.



Rysunek 4.7. Przepływ sterowania przy tworzeniu widoku diagramu (dla czytelności uwzględnione zostały tylko klasy kontrolera - w metodach refresh odwołują się one do klas widoku)

Ustawianie domyślnych wartości atrybutów
Tworząc nowy komponent edytor odwołuje się z zapytaniem o standardową wartość jaką należy ustawić dla poszczególnych atrybutów do jednej klasy z następującego zestawu:

SingleValueValidator - odpowiada za wartości unikalne w obrębie deskryptora CAD,
SingleValuePerComponentValidator - odpowiada za wartości unikalne w obrębie danego typu komponentu,
MultipleValueValidator - odpowiada za atrybuty, które jako wartość mogą posiadać zestaw wartości i wymagane jest, aby zestawy te były unikalne w obrębie deskryptora CAD,
NonValidatedValues - odpowiada za generowanie wartości atrybutów, dla których nie ma żadnych ograniczeń co do powielania.

Przyjęte zostały różne polityki generowania wartości domyślnej w zależności od parametru. Omówienie wspomnianych polityk zawiera tabela 4.1 (odwzorowanie stałych na atrybuty znaczników deskryptora *CAD* zostało opisane w *Dodatku A*).

Wspomniany w tabeli *licznik* mierzy liczbę wygenerowanych wartości domyślnych dla danego typu komponentu - zwykle jest ona równa liczbie instancji komponentu.

Aby nie przytłaczać użytkownika edytora ogromną liczbą parametrów konfiguracyjnych prezentowany jest w widoku właściwości (*Properties View*) jedynie ich podzbiór (ukryte wartości uzyskują standardowe wartości automatycznie):

- COMPONENTASSEMBLY_ID,
- COMPONENTFILE_FILE,
- COMPONENTPROPERTIES_FILE,
- COMPONENTIMPLREF_IDREF,
- COMPONENTINSTANTIATION_ID,
- REGISTERCOMPONENT_NAME,
- ACTIVATION_NAME,
- INSTALLATION_NAME,
- DESTINATION_NODE,
- REGISTERHOMEWITHNAMING_NAME,
- REGISTERWITHHOMEFINDER_NAME.

Pole	Wartość domyślna
COMPONENTASSEMBLY_ID	wartość wspólna dla wszystkich komponentów, brana aktualna wartość
COMPONENTINSTANTIATION_ID	typ komponentu + '#' + numer instancji
COMPONENTFILE_ID	typu absolutny komponentu +'CSD_'+licznik
COMPONENTFILE_FILE	'archives/' + typ komponentu + '.car'
COMPONENTFILE_LINK	pusty łańcuch
COMPONENTPROPERTIES_FILE	'META-INF/'+ typ komponentu+'cpf'
COMPONENTPROPERTIES_LINK	pusty łańcuch
COMPONENTIMPLREF_IDREF	typ komponentu + 'Impl'
REGISTERCOMPONENT_NAME	typ absolutny komponentu
REGISTERHOMEWITHNAMING_NAME	typ absolutny komponentu + 'Home' + licznik
REGISTERWITHHOMEFINDER_NAME	j.w.
INSTALLATION_NAME	pusty łańcuch albo ostatnio wybrane dla typu komponentu miejsce alokacji (dla pierwszego komponentu jest to domyślne miejsce alokacji), zgodnie z DTD ustawiane jest albo DESTINATION_NODE albo INSTALLATION_NAME i ACTIVATION_NAME
ACTIVATION_NAME	j.w.
DESTINATION_NODE	j.w.
INSTALLATION_TYPE	'componentinstallation'
ACTIVATION_TYPE,	'componentserver'

Tablica 4.1. Polityki generowania domyślnych wartości

4.2.2. Intuicyjna reprezentacja miejsc alokacji komponentów (*Destination*)

Opis realizacji tego wymagania zacznijmy od przedstawienia wprowadzonego pojęcia, którym będziemy się posługiwać. W środowiksu *CCM* istnieje pojęcie o angielskiej nazwie *Destination,* które w tym dokumencie będziemy tłumaczyć jako *miejsce alokacji.* Miejscem alokacji może być:

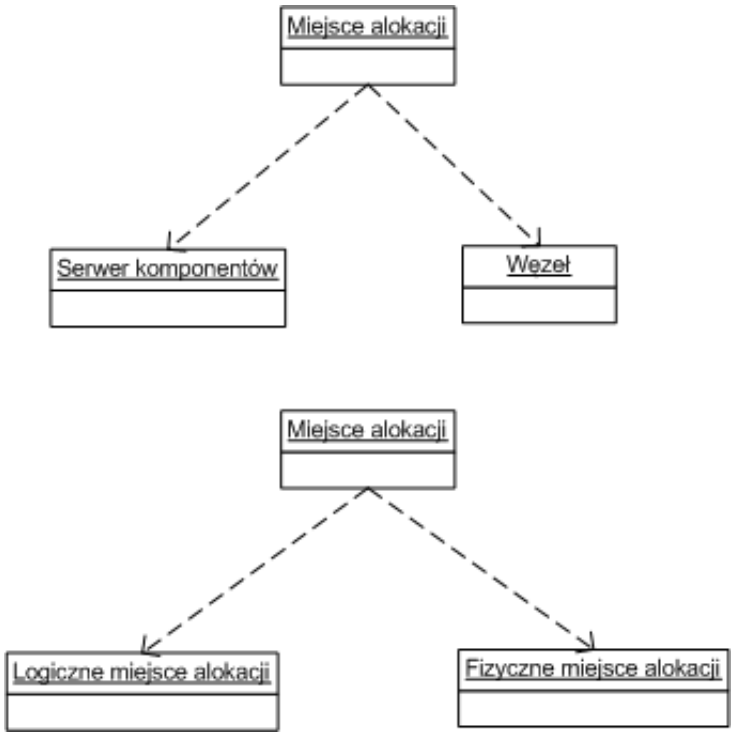
- węzeł (*node*),
- serwer komponentów (*component server*).

Oprócz tego miejsce alokacji może być:

- logiczne - logiczne miejsce alokacji pełni rolę agregatora dla komponentów.

- fizyczne - fizyczne miejsce alokacji to fizyczny węzeł lub serwer komponentów.

Rysunek 4.8 przedstawia powyższe zależności.



Rysunek 4.8. Miejsca alokacji

Zgodnie z wymaganiami zaimplementowaliśmy funkcjonalność logicznych miejsc alokacji. Zgonie z powyższym stwierdzeniem logiczne miejsce to serwer lub węzeł, który nie ma swojego odzwierciedlenia w rzeczywistym środowisku. Pełni funkcję agregatora dla komponentów. Jest ono używane do grupowania komponentów w przypadku, gdy nie mamy jeszcze wczytanych informacji o fizycznych miejscach działających w środowisku. Po dokonaniu grupowania, gdy mamy już dane o włączonych fizycznych serwerach, można w łatwy sposób dokonać odwzorowania logicznego na fizyczne miejsce alokacji.

Zaimplementowane właściwości:

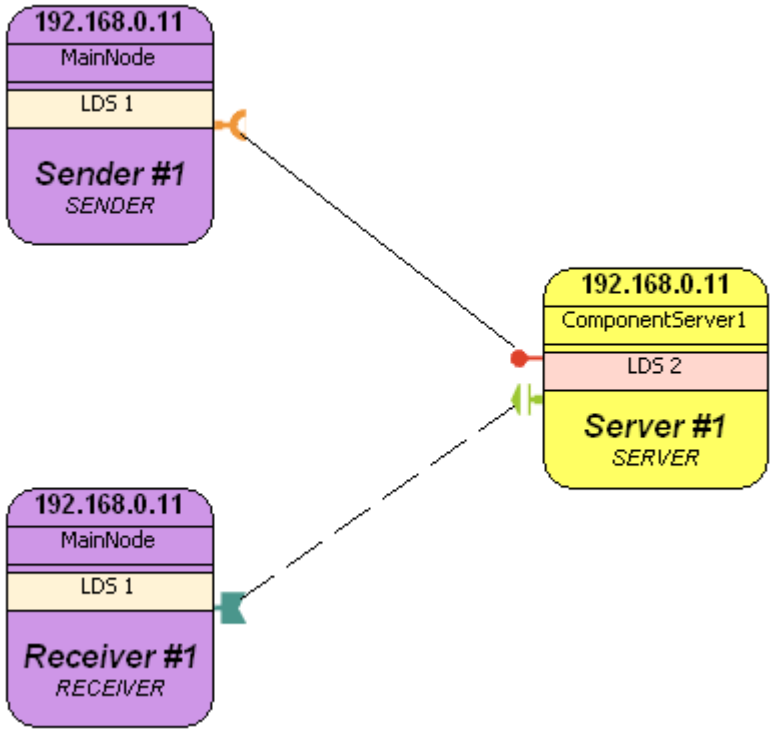
- podczas inicjalizacji programu lista fizycznych miejsc nie zawiera żadnych miejsc, ponieważ żadne fizyczne serwery nie są jeszcze wczytane. Lista ta zawiera jedynie wartość *unmapped,* informująca, że nie ma dostępnych informacji o fizycznych miejscach. Wszystkie komponenty mają przypisaną właśnie tę wartość jako fizyczne miejsce.
- po połączeniu do serwisu nazw i załadowaniu informacji o fizycznych serwerach oraz węzłach otrzymujemy listę fizycznych miejsc. Wybrane zostaje **domyślne fizyczne miejsce**, który zastępuje wartość *unmapped.*
- można dodawać kolejne logiczne miejsca. Domyślna ilość wynosi 3, a maksymalna 10. W celu zwiększenia ilości logicznych serwerów należy wcisnąć przycisk *Add*

destination, natomiast przycisk *Delete Destination* powoduje usunięcie ostatniego logicznego miejsca. Przyciski te znajdują się w dolnej części panelu *Destinations Mapping View*. Usunięcie logicznego miejsca jest możliwe pod warunkiem, że żaden komponent nie jest przypisany do niego.

- z każdym miejscem fizycznym jak i logicznym stowarzyszony jest inny kolor.
- każdy tworzony komponent otrzymuje domyślne wartości dla logicznego i fizycznego miejsca alokacji. Domyślne logiczne miejsce posiada nazwę *default*, natomiast domyślnym fizycznym miejscem jest pierwsze z listy fizycznych miejsc.
- ponieważ przynależność do danego miejsca wiąże się jedynie z kolorem komponentu nie ma problemu z istnieniem wielu instancji danego miejsca.
- zaimplementowane rozwiązanie nigdy nie wymaga zmiany położenia stworzonych wcześniej obiektów podczas procesu przypisywania komponentom miejsca alokacji.

Wizualizacja serwerów komponentów poprzez kolorowanie komponentów

Przyjęte rozwiązanie prezentowania miejsc alokacji wykorzystuje kolory elementów komponentu do wskazania do jakiego miejsca alokacji dany komponent należy. Tło pola etykiet przedstawiających logiczne i fizyczne miejsce posiada kolor związany z danym miejscem. W zależności od wybranego widoku (widoki zostaną opisane w 4.2.3) cały komponent przyjmuje kolor odpowiedniego logicznego albo fizycznego miejsca alokacji. Dzięki zastosowaniu tego rozwiązania użytkownik w błyskawiczny sposób może dostrzec, które komponenty należą do tego samego miejsca. Zrealizowany sposób prezentacji odwzorowywania przedstawia rysunek 4.9.



Rysunek 4.9. Odwzorowanie przynależności komponentu do logicznego i fizycznego miejsca alokacji

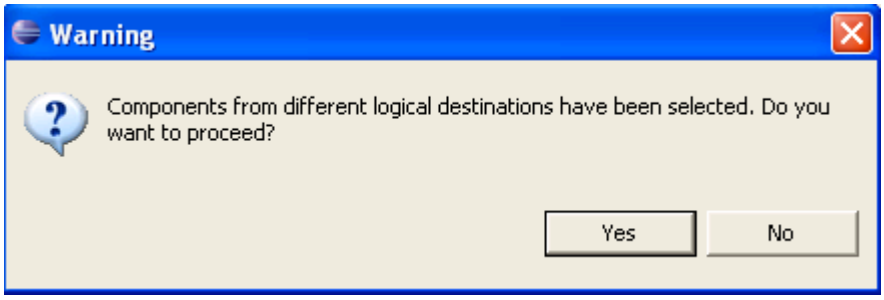
Ustawianie logicznych i fizycznych serwerów komponentów dla danego komponentu możliwe jest na 3 sposoby:

- **użycie właściwości (*properties*) komponentu** - okno *properties* dla komponentu posiada między innymi możliwość ustawienia logicznego i fizycznego miejsca alokacji (pola *Logical Destination* i *Physical Destination*). Wybór odbywa się poprzez zmianę wartości w polu typu *ComboBox*. Zmiana wartości powoduje zmiany kolorów i etykiet komponentu (zmiany te nie są widoczne na widoku komponentów).
- **użycie narzędzia select** - zgodnie z wymaganiem wygodnego i szybkiego przypisywania komponentu do miejsca alokacji stworzyliśmy możliwość wykorzystania palety z obiektami miejsc. Wybierając z palety ikonkę reprezentującą odpowiednie miejsce alokacji możemy poprzez kliknięcie na wybrany komponent zmienić jego odwzorowanie. Istnieje także możliwość grupowego przypisania; w tym celu wystarczy otoczyć wybrane komponenty po wcześniejszym kliknięciu ikonki wybranego miejsca alokacji. Narzędzie *select* dla miejsc alokacji pojawia się na palecie diagramu w postaci ikonki zaprezentowanej na rysunku 4.10

Rysunek 4.10. Ikona narzędzia służącego do oznaczania jako fizyczny serwer komponentów o nazwie *ComponentServer1*

  Kolory logicznych miejsc są jasne, natomiast fizyczne miejsca stowarzyszone są z kolorami bardziej intensywnymi.
- **użycie panelu *Destinations Mapping View*** - panel ten umożliwia dokonanie odwzorowania logicznych na fizyczne miejsca. Zawiera on listę logicznych serwerów, do których można przypisać dostępne miejsca fizyczne. Sposób ten nie jest więc bezpośrednim przypisaniem miejsca wybranemu komponentów. Dokonywane zmiany zachodzą pośrednio - zmiana przypisania fizycznegeo miejsca dla danego logicznego miejsca skutkuje zmianą przypisanego fizycznego miejsca dla wszystkich komponentów mających przypisany wybrany logiczny serwer.

W przypadku gdy użytkownik zaznaczy komponenty znajdujące się w różnych logicznych miejscach i zmieni przypisane fizyczne miejsce dla danego logicznego miejsca zostaje wyświetlony komunikat zaprezentowany na rysunku 4.11.



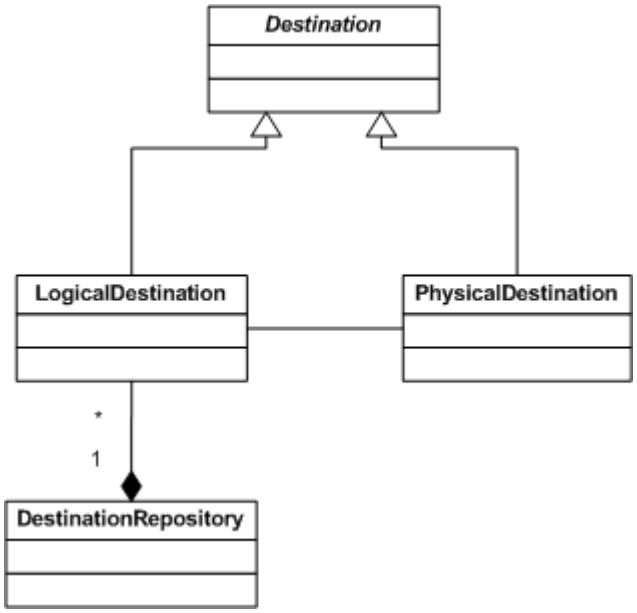
Rysunek 4.11. Ostrzeżenie o zaznaczonych komponentach z różnych logicznych miejsc

Komunikat ten jest potrzebny ze względu na istnienie jednego fizycznego miejsca przypisanego do logicznego miejsca alokacji. Komunikat ten powoduje, że nieświadomy użytkownik nie będzie zaskoczony reakcją systemu na taką zmianę. Jeśli użytkownik potwierdzi wykonanie operacji nastąpi przypisanie wybranego fizycznego miejsca do każdego komponentu należącego do jednego z zaznaczonych logicznych miejsc.

Implementując koncepcję reprezentacji rozmieszczenia poprzez kolorowanie komponentów wprowadziliśmy następujące klasy:

1. Klasa *Destination.*
   Klasa bazowa dla klas *LogicalDestination* oraz *PhysicalDestination* zapewniająca wspólną funkcjonalność tych klas. Posiada następujące atrybuty:
   - nazwa,
   - kolor,
   - id - wykorzystywany przy generowaniu koloru.
2. Klasa *LogicalDestination.*
   Klasa ta zawiera dane związane z logicznym miejscem alokacji; rozszerza klasę *Destination* o następujące atrybuty:
   - przypisane fizyczne miejsce alokacji,
   - referencja obiektu klasy *DestinationRepository.*
3. Klasa *PhysicalDestination.*
   Klasa ta zawiera dane związane z fizycznym miejscem alokacji; rozszerza klasę *Destination* o następujące atrybuty:
   - host - adres ip komputera, na którym serwer jest uruchomiony,
   - typ - serwer komponentów lub węzeł.
4. Klasa *DestinationRepository.*
   Klasa zawiera listy logicznych i fizycznych serwerów komponentów oraz metody dostępowe do agregowanych obiektów. Tutaj też następuje definiowanie i przypisywanie kolorów do poszczególnych miejsc alokacji.

Zależności pomiędzy wymienionymi powyżej klasami prezentuje rysunek 4.12.



Rysunek 4.12. Diagram zależności pomiędzy klasami

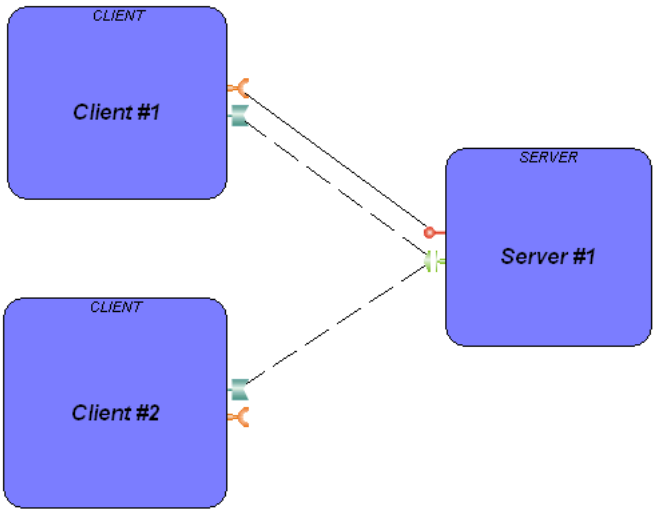
Wszystkie klasy znajdują się w pakiecie: *pl.go.die.editor.model.*

Przypisanie komponentu do miejsca alokacji realizowane jest przez komendę *AssignDestinationCommand,* zaś zmiana odwzorowania logicznego miejsca rozmieszczenia na fizyczne przy użyciu *EditLogicalDestCommand.*

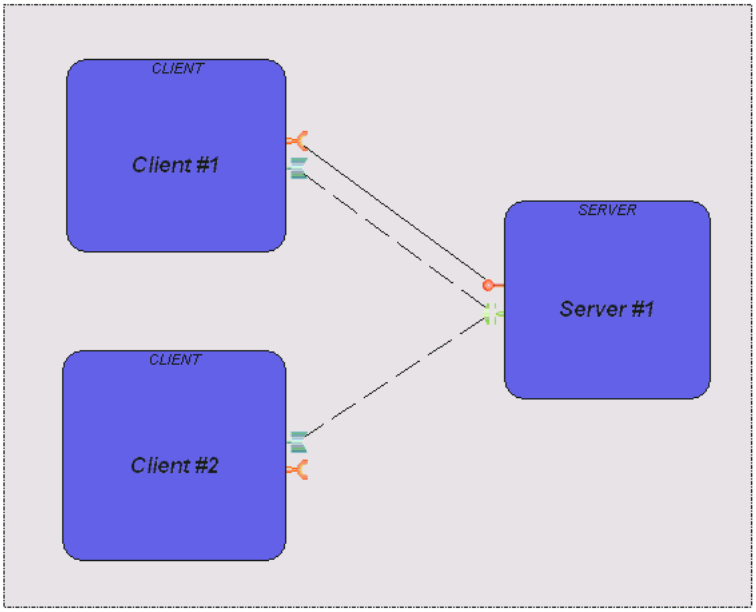
Alternatywny sposób wizualizacji serwerów komponentów

Podczas tworzenia aplikacji *CGE* pojawił się pomysł prezentowania na diagramie miejsc alokacji jako kontenerów mogących zawierać komponenty. Do wcześniej stworzonych serwerów można było przerzucać komponenty, które miały być umieszczone w serwerze reprezentowanym przez kontener. Rozwiązanie takie posiada duży plus, jakim jest jego intuicyjność dla użytkownika. Struktura aplikacji staje się bardziej czytelna. Aby jeszcze bardziej udoskonalić to rozwiązanie dodaliśmy możliwość przypisania miejsca alokacji wcześniej stworzonym komponentom. Funkcjonalność tą zobrazują następujące kroki:

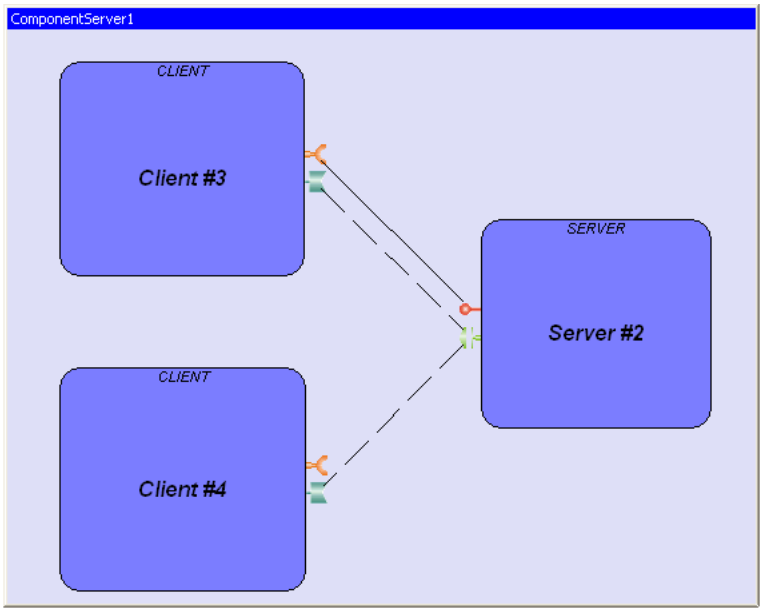
1. Tworzymy komponenty - rysunek 4.13.
2. Rysujemy kontener serwera komponentów otaczającego istniejące komponenty - rysunek 4.14.
3. Serwer komponentowy zawiera obiekty przedstawiające komponenty, które znajdowały się w obszarze rysowanego serwera - rysunek 4.15.



Rysunek 4.13. Narysowane komponenty



Rysunek 4.14. Tworzymy serwer komponentów



Rysunek 4.15. Serwer komponentów zawiera komponenty, które znajdowały się w obszarze rysowanego serwera

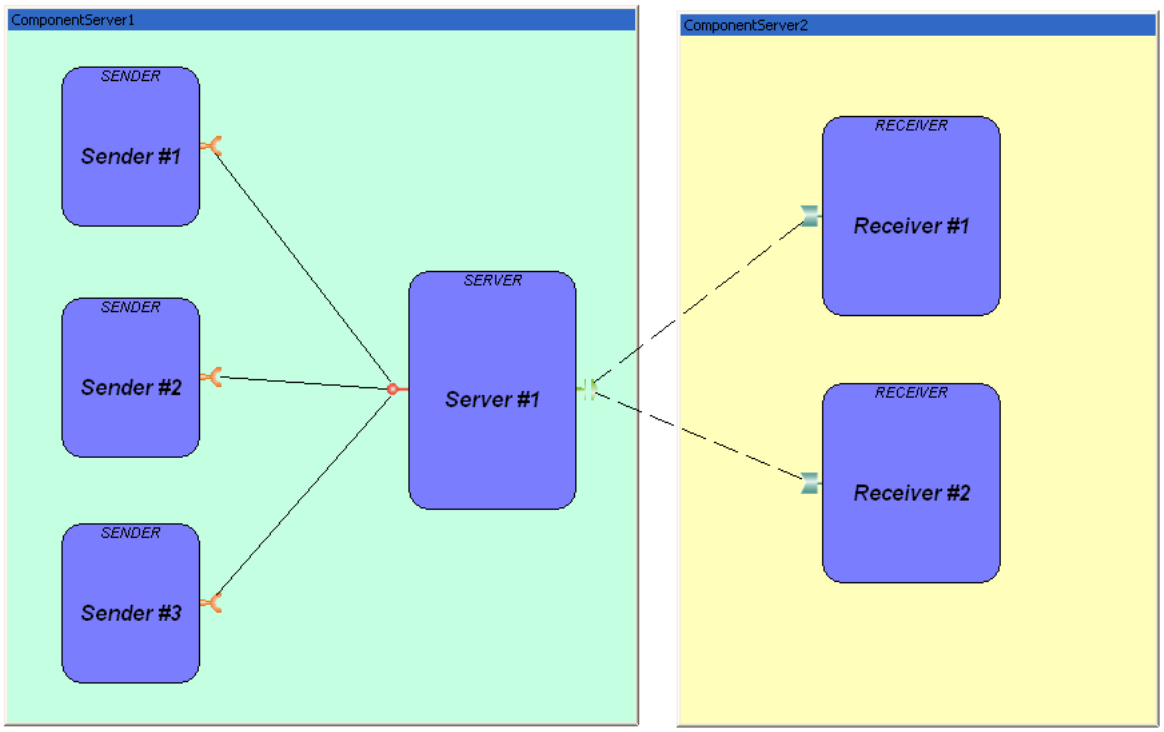
Wprowadzenie takiego procesu alokacji wymagało:

- implementacji dwóch polityk (realizujących role *EditPolicy.LAYOUT*, *EditPolicy.CONTAINER*) dla składowej kontrolera zajmującej się miejscem alokacji,
- implementacji komend tworzonych przez polityki w odpowiedzi na żądania użytkownika, a dotyczących dodania albo usunięcia komponentu do/z miejsca alokacji,
- oprócz nowych elementów należało również zmodyfikować już istniejące w szczególności związane z diagramem (umieszczenie komponentu w miejscu-kontenerze równoznaczne jest z usunięciem go z diagramu).

Po wprowadzeniu miejsc alokacji reprezentowanych za pomocą kontenerów interfejs użytkownika wyglądał jak na rysunku 4.16.

To intuicyjne rozwiązanie nie było pozbawione wad. Niektóre z nich były na tyle poważne, że użycie tej techniki stało się niemożliwe:

- w rozwiązaniu "kontenerowym" podzielenie serwera komponentów na mniejsze części było znacznie utrudnione.
- problem pojawiał się także podczas zmiany rozmiaru kontenera, ponieważ zawarte w nim komponenty mogły przestać być widoczne.
- dodatkowo każdy z serwerów komponentów powinien dać się narysować w wielu instancjach - tak aby możliwe było przypisanie odległych od siebie na diagramie komponentów do tego samego serwera komponentów. Rozwiązanie takie było mało intuicyjne i trudne w realizacji w środowisku *GEF*.
- istniała także obawa, że podczas przełączania widoków (omówionych w kolejnym podrozdziale) stworzone komponenty mogą zostać narysowane pod kontenerem serwera, który na danym widoku nie będzie wyświetlany. W rezultacie kontener zasłoniłby komponent na innych widokach.



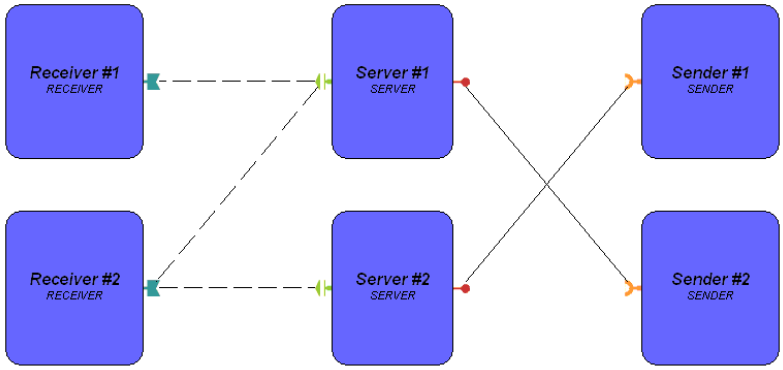
Rysunek 4.16. Interfejs użytkownika uwzględniający rysowanie kontenerów

- istniały scenariusze zmiany przypisania serwerów komponentów, w których nie można było obejść się bez przemieszczania komponentów. Przesuwanie komponentów podczas przypisywania do serwerów komponentów jest niezgodne z postawionymi aplikacji wymaganiami.

4.2.3. Wielopoziomowa prezentacja diagramu

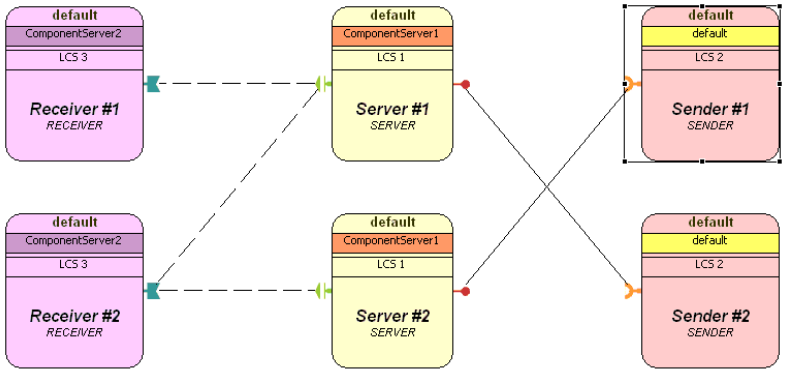
Aby korzystanie z aplikacji było bardziej wygodne wprowadziliśmy trzy widoki prezentujące ten sam diagram, lecz uwypuklające inne jego aspekty:

1. **Widok komponentów**
   Widok ten przedstawia jedynie komponenty i połączenia pomiędzy nimi. W widoku tym można modyfikować diagram jedynie na poziomie komponentów. Logiczne i fizyczne miejsca alokacji nie są rysowane.
   Dzięki wprowadzeniu tego widoku mamy możliwość szybkiego stworzenia diagramu bez potrzeby przypisywania poszczególnych komponentów do serwerów komponentów. Każdy z komponentów jest przypisany do domyślnego miejsca alokacji (opis domyślnych miejsc alokacji znajduje się 4.2.2 na 53 stronie). Odczyt i zmiana przypisania do serwerów komponentów jest jednak możliwa. Można do tego celu wykorzystać okno właściwości (*properties*) lub narzędzie typu *select* znajdujące się na palecie.



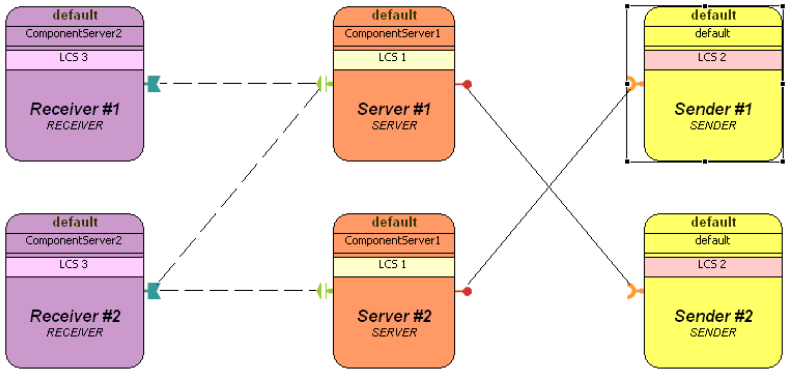
Rysunek 4.17. Widok komponentów

2. **Widok logicznych miejsc alokacji**
   Widok ten przedstawia komponenty, połączenia pomiędzy nimi, przy czym kolor komponentu odzwierciedla przynależność do **logicznego miejsca alokacji**. Komponent posiada trzy etykiety, które prezentują wybrany logiczny i fizyczny serwer komponentów oraz adres IP hosta, na którym uruchomiony jest fizyczny serwer. Kolor każdej z etykiet związany jest z miejscem alokacji, które prezentuje. W widoku tym mamy możliwość przypisywania fizycznych i logicznych serwerów komponentów oraz funkcjonalność widoku komponentów, czyli łączenie i tworzenie komponentów.



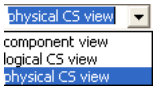
Rysunek 4.18. Widok logicznych miejsc alokacji

3. **Widok fizycznych miejsc alokacji**
   Widok ten przedstawia komponenty, połączenia pomiędzy nimi, przy czym kolor komponentu odzwierciedla przynależność do **fizycznego miejsca alokacji.** Podobnie jak w poprzednim przypadku komponent posiada trzy pola, które zawierają etykiety prezentujące wybrany logiczny i fizyczny serwer komponentów oraz adres IP hosta, na którym uruchomiony jest fizyczny serwer. Kolor każdej z etykiet związany jest z miejscem alokacji, które prezentuje. W widoku tym mamy możliwość przypisywania fizycznych i logicznych serwerów komponentów oraz funkcjonalność widoku komponentów, czyli łączenie i tworzenie komponentów.



Rysunek 4.19. Widok fizycznych miejsc alokacji

Przełączanie pomiędzy widokami odbywa się poprzez wybór widoku z pola wyboru znajdującego się w górnej części okna środowiska *Eclipse.*



Rysunek 4.20. Przełączanie widoków

Zmiana widoku modyfikuje sposób prezentowania diagramu - następuje zmiana wyglądu komponentów.

Następujące klasy powstały w systemie, w celu obsłużenia akcji związanych z widokami i ich obsługą:

ViewComboContributionItem - klasa zajmująca się wyświetlaniem kontrolki typu *ComboBox.* Wyświetlanymi etykietami są nazwy widoków. Klasa ta dziedziczy po abstrakcyjnej klasie biblioteki *SWT* o nazwie *ContributionItem,* która zapewnia podstawową funkcjonalność dla kontrolek typu rozwijanego menu czy paska narzędzi implementując interfejs *IContributionItem.* Obiekty klas dziedziczących po tym interfejsie są zarządzane przez menedżera (*contribution manager*), dzięki czemu korzystanie z kontrolek tego typu jest wygodne w użyciu przez programistę. Klasa znajduje się w pakiecie *pl.go.die.editor.actionbuttons.*

ChangeViewCommand - klasa reprezentująca komendę zmiany widoku. Klasa znajduje się w pakiecie: *pl.go.die.editor.model.commands.*

Następujące klasy uległy modyfikacjom potrzebnym do obsługi widoków:

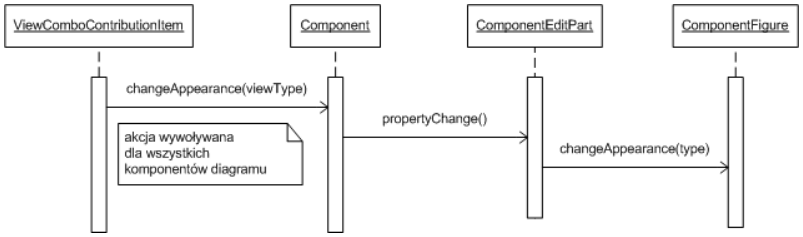
CCMDiagram - klasa reprezentująca tworzony diagram. Udostępnia ona metodę *getCurrentView()* umożliwiającą pobranie aktualnego widoku dla wyświetlanego diagramu.

Component - klasa, reprezentująca model komponentu diagramu, posiada metodę *changeAppearance,* której wywołanie powoduje poinformowanie kontrolera związanego z daną instancją klasy, że nastąpiła zmiana widoku.

ComponentEditPart - klasa kontrolera komponentu. Kontroler przekazuje otrzymaną informację od obiektu klasy *Component* do obiektu widoku związanego z danym obiektem kontrolera.

ComponentFigure - klasa widoku komponentu. Gdy obiekt tej klasy otrzyma żądanie zmiany wyglądu, zmienia kolor tła komponentu prezentowanego na diagramie na kolor związany z żądanym (logicznym lub fizycznym) miejscem alokacji lub na domyślny kolor w przypadku przełączenia na widok komponentów.

Opisane powyżej akcje zaprezentowane są na rysunku 4.21.



Rysunek 4.21. Diagram sekwencji - zmiana widoku

4.2.4. Różne sposoby prezentacji diagramu

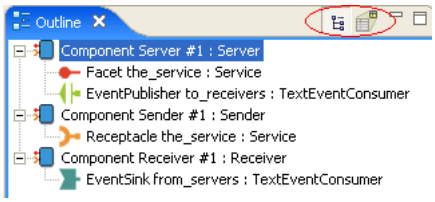
Biblioteka *GEF* wspiera obsługę standardowych akcji użytkownika związanych z tworzeniem diagramu. Nawigacja w przypadku tworzenia złożonego modelu jest ułatwiona dzięki automatycznemu pojawianiu się suwaków, w przypadku gdy rysowane elementy nie mieszczą się na ekranie. Dodatkowym ułatwieniem jest możliwość wyświetlenia diagramu w trybie pełnoekranowym. Aktywacja tego trybu dokonywana jest poprzez dwukrotne kliknięcie paska tytułowego tworzonego diagramu i skutkuje zniknięciem wszystkich ramek edytora poza tą, zawierającą diagram.

Z głównym oknem edytora, gdzie prezentowany jest diagram, stowarzyszone są dodatkowe panele prezentujące w alternatywny sposób tworzony diagram:

drzewo - zapewnia prezentowanie modelu w postaci drzewa. Taki sposób przedstawienia diagramu dobrze opisuje jego strukturę. Poprzez kliknięcie na wybrany element drzewa mamy możliwość odnalezienia na diagramie w głównym oknie edytora zaznaczanego obiektu. Jeśli zaznaczony obiekt znajduje się na obszarze aktualnie nie wyświetlanym to diagram zostanie przesunięty w taki sposób, aby wybrany obiekt był widoczny.

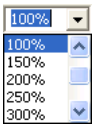
miniatura - zapewnia prezentowanie miniatury modelu, co w przypadku dużych diagramów ułatwia nawigację.

Panele te są dostępne z poziomu standardowego w środowisku *Eclipse* widoku o nazwie ***Outline.*** Przełączanie pomiędzy drzewem a miniaturą odbywa się poprzez kliknięcie na wybraną ikonkę (na rysunku zakreślone czerwoną elipsą) znajdującą się w górnej części widoku *Outline* (rysunek 4.22).



Rysunek 4.22. Panel *Outline*

W przypadku bardzo rozbudowanych modeli tworzonych za pomocą edytora może dojść do sytuacji, w której użytkownik nie będzie w stanie ogarnąć wzrokiem wszystkich komponentów. Dostępne są suwaki umożliwiające przesuwanie widoku w odpowiednie miejsce diagramu. Jednak korzystanie z suwaków jest bardzo uciążliwe. Dlatego też wprowadziliśmy możliwość korzystania z funkcji przybliżania i oddalania (*zoom in/out*). Z funkcji tej można skorzystać wybierając żądaną wartość z przycisku typu *ComboBox* znajdującego się w górnej części edytora lub poprzez użycie rolki myszki wraz z jednoczesnym trzymaniem przycisku *CTRL* na klawiaturze. Przycisk wyboru przybliżenia prezentowany jest na rysunku 4.23



Rysunek 4.23. Przycisk wyboru przybliżenia

Dodatkową funkcją stowarzyszoną z przybliżaniem jest wyświetlanie na panelu miniatury diagramu aktualnego widocznego obrazu (niebieksa ramka posiadająca możliwość przesuwania, co skutkuje przesunięciem obrazu w głównym oknie edytora).

Funkcjonalność drzewa w edytorze zapewniają klasy kontrolera (pakiet *pl.go.die.editor.controller*):

CCMDiagramTreeEditPart - klasa umożliwiająca prezentowanie diagramu w postaci drzewa,

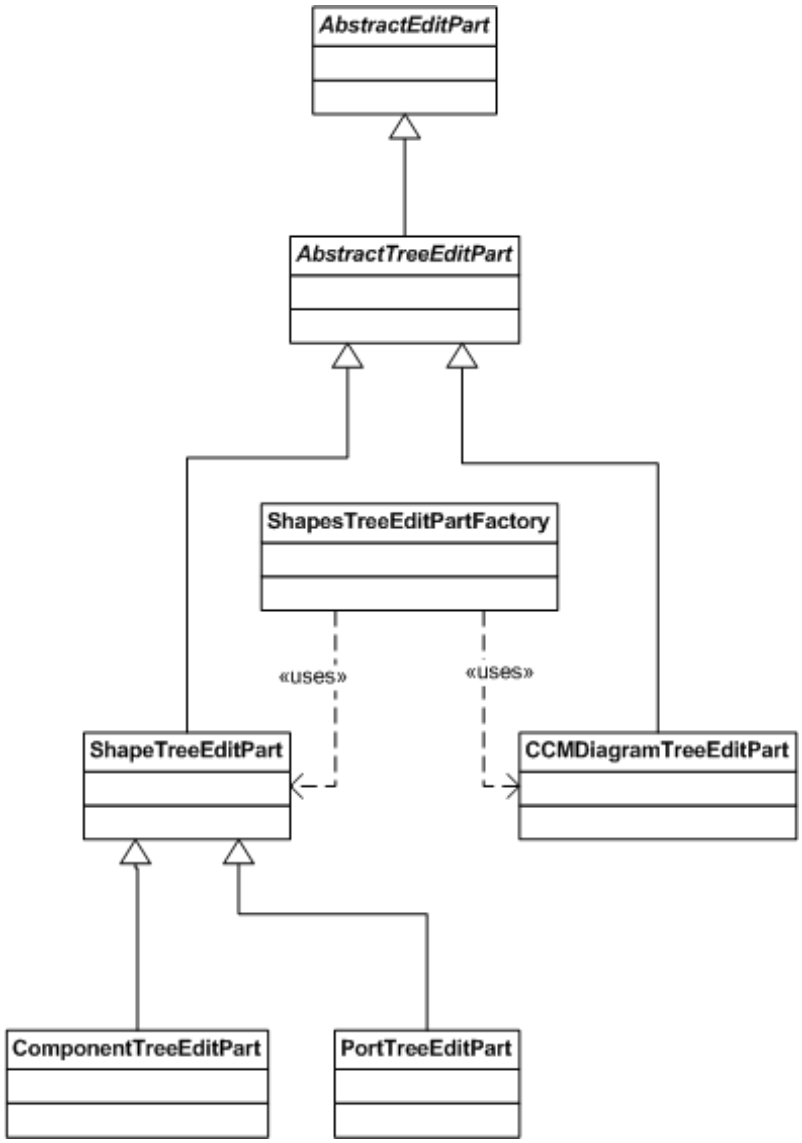
ShapesTreeEditPartFactory - mapuje obiekty modelu na ich odpowiedniki w kontrolerze obsługujące poszczególne elementy drzewa,

ShapeTreeEditPart - klasa bazowa dla elementów wyświetlanych na drzewie zapewniająca podstawową funkcjonalność związaną z prezentacją obiektu na drzewie,

ComponentTreeEditPart - klasa obsługująca prezentowanie obiektów modelu komponentu na drzewie,

PortTreeEditPart - prezentuje obiekty modelu portu na drzewie.

Diagram na rysunku 4.24 prezentuje zależności pomiędzy poszczególnymi klasami.



Rysunek 4.24. Zależności pomiędzy klasami reprezentującymi funkcjonalność panelu *Outline*

Do obsługi funkcjonalności przybliżania/oddalania używamy klas:

CADActionBarContributor - zarządza akcjami wywoływanymi z paska narzędzi. Rejestruje obiekty klasy *ZoomComboContributionItem* oraz *ViewComboContributionItem* (opisane w 4.2.3). Klasa ta należy do pakietu: *pl.go.die.editor.actionbuttons.*

ZoomComboContributionItem - klasa kontrolki zmiany stopnia przybliżenia. Klasa ta należy do następującego pakietu biblioteki *GEF: org.eclipse.gef.ui.actions.*

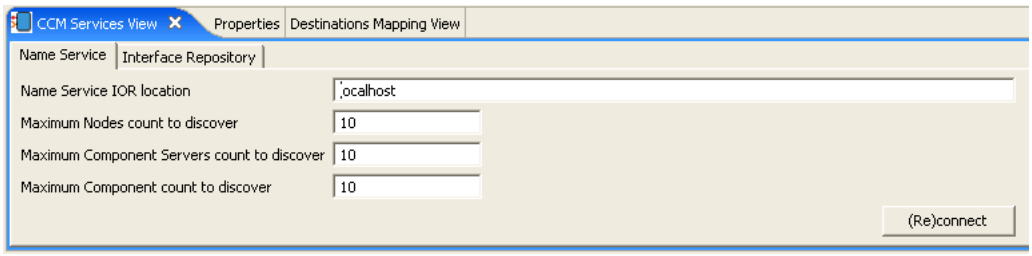
ZoomManager - klasa ta zapewnia funkcjonalność przybliżania i oddalania w edytorze. Należy do pakietu: *org.eclipse.gef.editparts.*

4.2.5. Pobranie definicji komponentów i miejsc alokacji

Edytor *CGE* umożliwia pobranie informacji o miejscach alokacji z serwisu nazw (*Name Service*) oraz o komponentach z repozytorium interfejsów (*Interface Respository*). Miejscami alokacji są:

- serwery komponentowe (*Component Server*),
- węzły (*Node*).

Połączenie do serwisów nazw odbywa się z wykorzystaniem widoku *CCM Services View*. Widok ten posiada dwie zakładki. Zakładka umożliwiająca połączenie z serwisem nazw posiada nazwę *Name Service* i jest zaprezentowana na rysunku 4.25.



Rysunek 4.25. Zakładka *Name Service*

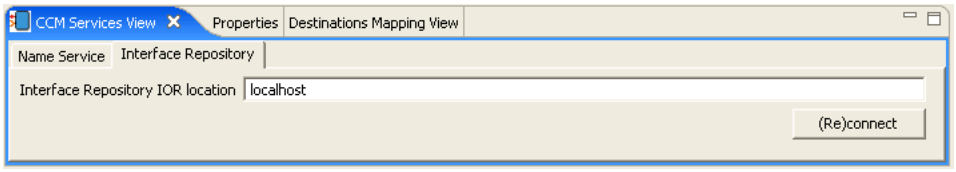
Zakładka ta umożliwia:

- wprowadzenie adresu, pod którym znajduje się referencja do serwisu (*IOR - Interoperable Object Reference*). Dopuszczalne są następujące formaty adresu:
  1. ***localhost*** - wpisana fraza *localhost* powoduje, że referencja IOR jest poszukiwana w domyślnym katalogu środowiska *OpenCCM*: *OpenCCM_ CONFIG_ DIR*. Szukana referencja znajduje się w pliku *NameService.IOR*.
  2. ***corbaloc*** - fraza rozpoczynająca się od słowa *corbaloc:* oznacza adres serwisu lokalizacji referencji *IOR corbaloc*. Adres ten zawiera adres IP serwisu, port na którym włączona jest usługa oraz nazwę serwisu, do którego chcemy się połączyć. Przykładowy adres ma postać:

     corbaloc:iiop:192.168.0.11:2001/NameService.
  3. ***http*** - środowisko *OpenCCM* zawiera narzędzie o nazwie *comanche,* które uruchamiamy za pomocą polecenia *comanche_ start.* Program ten udostępnia referencję do serwisu poprzez protokół *http*. Adres tego formatu zawiera następujące informacje: adres *IP* serwisu, port na którym włączona jest usługa oraz nazwę pliku *IOR,* który chcemy pobrać.
- określenie maksymalnej ilości węzłów, które mają zostać wczytane (*Maximum Nodes count to discover*).
- określenie maksymalnej ilości serwerów komponentowych, które mają zostać wczytane (*Maximum Component Server count to discover*).
- określenie maksymalnej ilości osadzonych komponentów, które mają zostać wczytane (*Maximum Component count to discover*).

Przycisk *(Re)connect* umożliwia połączenie do serwisu nazw z wykorzystaniem wprowadzonego adresu oraz odświeżenie informacji o wcześniej wczytanych miejscach alokacji. Pobranie danych z serwisu objawia się uzupełnieniem palety o wczytane serwery komponentowe i węzły oraz pojawieniem się komunikatu potwierdzającego połączenie. W przypadku niepowodzenia wyświetlony zostaje stosowny komunikat.

Druga zakładka widoku *CCM Services View* pozwala na połączenie do repozytorium interfejsów. Zakładka *Interface Respository* przedstawiona jest na rysunku 4.26.



Rysunek 4.26. Zakładka *Interface Repository*

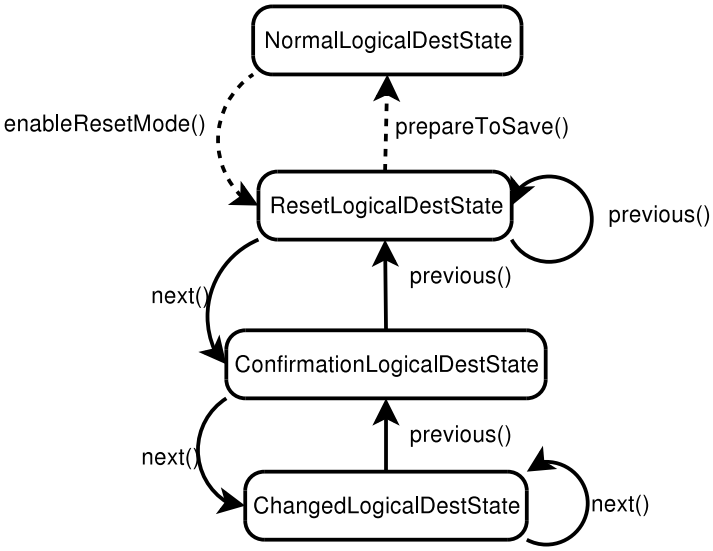
Zakładka ta umożliwia wprowadzenie adresu, pod którym znajduje się referencja do serwisu (*IOR - Interoperable Object Reference*). Dopuszczalne są następujące formaty adresu:

1. ***localhost*** - wpisana fraza *localhost* powoduje, że referencja IOR jest poszukiwana w domyślnym katalogu środowiska *OpenCCM*: *OpenCCM_ CONFIG_ DIR*. Szukana referencja znajduje się w pliku *IR3.IOR*.
2. ***http*** - środowisko *OpenCCM* zawiera narzędzie o nazwie *comanche,* które uruchamiamy za pomocą polecenia *comanche_ start*. Program ten udostępnia referencję do serwisu poprzez protokół *http*. Adres tego formatu zawiera następujące informacje: adres IP serwisu, port na którym włączona jest usługa oraz nazwę pliku *IOR*, który chcemy pobrać.

Przycisk *(Re)connect* umożliwia połączenie do repozytorium interfejsów z wykorzystaniem wprowadzonego adresu oraz odświeżenie informacji o wcześniej wczytanych komponentach. Pobranie danych z serwisu objawia się uzupełnieniem palety o wczytane komponenty oraz pojawieniem się komunikatu potwierdzającego połączenie. W przypadku niepowodzenia wyświetlony zostaje stosowny komunikat.

Podczas ponownego pobrania informacji z serwisu nazw może okazać się, że pewne miejsca alokacji nie są już dostępne. Domyślnie w takiej sytuacji odwzorowanie zmieniane jest na fizyczne miejsce alokacji powiązane z domyślnym logicznym miejscem alokacji. Aby zmiana ta została zapisana wymagane jest potwierdzenia ze strony użytkownika.

Logiczne miejsce alokacji może znajdować sie w jednym z czterech stanów. Do implementacji jego zachowania zastosowaliśmy wzorzec projektowy *State* do opisu stanu logicznego miejsca alokacji, co pokazuje rysunek 4.27.



Rysunek 4.27. Maszyna stanowa dla logicznego miejsca alokacji

NormalLogicalDestState jest to standardowy stan obiektu, z którego może przejść do *ResetLogicalDestState* w wyniku sytuacji wyjątkowej - fizyczne miejsce alokacji, na które mapuje się obiekt nie istnieje; do tego stanu obiekt powraca w czasie serializacji.

ResetLogicalDestState jest to stan, w którym wymagane jest od użytkownika potwierdzenie nowego domyślnego odwzorowania.

ConfirmationLogicalDestState obiekt znajduje się w tym stanie po zatwierdzeniu domyślnego odwzorowania przez użytkownika.

ChangedLogicalDestState jest to stan, w którym znajduje się obiekt po wykonaniu kolejnej zmiany odwzorowania.

Taka struktura maszyny stanowej pozwala na współpracę ze stosem komend dostarczanym przez środowisko *Eclipse*.

Do implementacji pobierania danych z serwisu nazw użyliśmy następujących klas:

- pakiet *pl.go.die.ccmapi*:
  NSDAO klasa zajmująca się połączeniem do serwisu nazw i pobraniem informacji o miejscach alokacji (realizacja wzorca projektowego *Data Access Object*[2]) .
  NodeManagersFoundException - wyjątek rzucany, w przypadku nie znalezienia obiektu *NodeManagers* w serwisie nazw.
  NSNotFound - wyjątek rzucany, w przypadku nieudanego połączenia do serwisu nazw.
  IRDAO - klasa zajmująca się pobraniem z repozytorium interfejsów danych o komponentach (realizacja wzorca projektowego *Data Access Object*).
  InvalidModulesNameException - wyjątek rzucany, w przypadku nie znalezienia wczytywanych modułów.

[2] dostęp do zewnętrznych danych realizowany jest za pomocą dedykowanego obiektu zapewniającego jednolity interfejs do komunikacji między aplikacją a źródłem danych

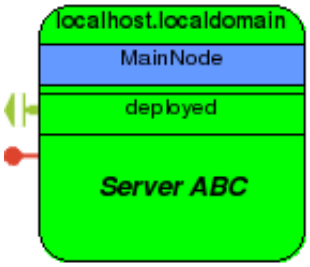
IRAccessException - wyjątek rzucany, w przypadku nieudanego połączenia do repozytorium.

- pakiet *pl.go.die.ccmapi.iorGetting*:
  IIORGetter - interfejs klas pomocniczych do pobierania referencji *IOR.*
  CorbalocIORGetter - klasa pomocnicza obsługująca pobranie referencji *IOR* za pomocą *corbaloc.*
  URL_IORGetter - klasa pomocnicza obsługująca pobranie referencji *IOR* za pomocą protokołu *http.*
  LocalIORGetter - klasa pomocnicza obsługująca pobranie referencji *IOR* z lokalnego systemu plików.
- pakiet *pl.go.die.editor.views*:
  CCMServicesView - klasa kontrolki zawierającej zakładki umożliwiające odczyt informacji z serwisu nazw i repozytorium interfejsów.

4.2.6. Wczytanie wcześniej rozmieszczonych komponentów

Edytor *CGE* został wyposażony w mechanizm wykrywania rozmieszczonych w środowisku komponentów. Podobnie jak w przypadku miejsc alokacji źródłem informacji jest *Naming Service.* Edytor wykorzystuje fakt rejestracji komponentów w tym serwisie. Po skonfigurowaniu parametrów dostępowych do serwisu *Naming Service* (proces opisany *w* 4.2.5), edytor nawiązując połączenie pobiera następujące informacje na temat aktualnie rozmieszczonych komponentów:

- nazwa pod jaką widoczny jest komponent w *Naming Service,*
- fizyczne miejsce alokacji, na którym komponent został osadzony,
- lista portów komponentu (dla portów rodzaju *event emitter i receptacle* wykrywane są już istniejące połączenia).

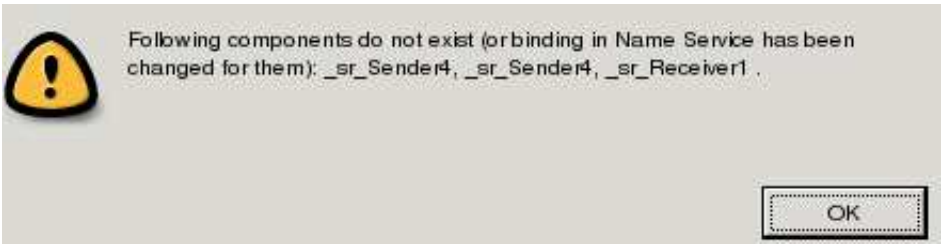


Rysunek 4.28. Widok osadzonego komponentu

Użytkownik może ograniczyć ilość komponentów do wykrycia. Osadzone komponenty traktowane są przez edytor *CGE* w sposób wyjątkowy:

- zarezerwowany został dla nich zielony kolor w widoku logicznych miejsc rozmieczenia, by łatwo było je rozróżnić (rysunek 4.28),
- nie jest możliwa dla nich zmiana miejsca alokacji,

- przy każdej próbie połączenia do *Naming Service* wykrywane i usuwane są komponenty, które przestały istnieć - o czym informowany jest użytkownik komunikatem (rysunek 4.29).



Rysunek 4.29. Ostrzeżenie o wykryciu w edytorze nieistniejących już komponentów

- osadzone komponenty ze względu na swoją ulotność nie są zapisywane wraz z diagramem,
- zapewniona jest możliwość łączenia portów komponentów rozmieszczonych i nieosadzonych (edytor nie zezwala na podłączenie się do portów już zajętych tzn. do portu *event emitter* albo *receptacle* posiadających już po jednym połączeniu - szczegóły w 4.2.7) - połączenia tego typu również nie są zapisywane z diagramem, ale są eksportowane do deskryptora CAD.

Opisana funkcjonalność została zrealizowana dzięki klasom*:*

NSDAO klasa dostępowa (realizacja wzorca projektowego *Data Access Object*) do *Naming Service,* umożliwia podłączenie do serwisu i uzyskanie definicji osadzonych komponentów,

DeployedComponent klasa reprezentuje w edytorze osadzony komponent,

CCMServicesView klasa rozszerzająca środowisko *Eclipse* o widok *CCM Services View* pozwalający na konfigurację połączenia do *NamingService.*

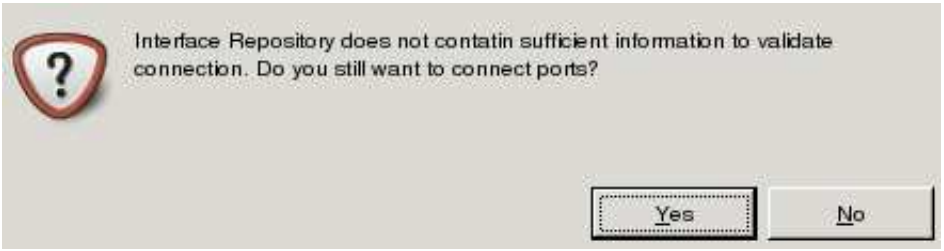
Szczegóły współpracy klas *CCMServicesView* oraz *NSDAO* zostaną omówione w 4.2.11.

4.2.7. Walidacja połączenia pomiędzy portami komponentów

Edytor CGE zapewnia walidację poprawności połączeń między portami. Proces ten jest dwuetapowy.

Etap 1: Sprawdzenie kompatybilności w oparciu o rodzaje łączonych portów Polega na weryfikacji reguł opisanych w 2.2.2, czyli sprawdzeniu czy łączone są dwa komplementarne porty synchroniczne albo asynchroniczne. Jest to warunek konieczny pozytywnego zakończenia walidacji.

Etap 2: Sprawdzenie kompatybilności w oparciu o typ łączonych portów Realizowany jest po przez sprawdzenie czy typy łączonych portów są kompatybilne. Zgodność typów definiuje się różnie w zależności od rodzaji portów:



Rysunek 4.30. Ostrzeżenie o łączeniu portów nieznanych typów

porty synchroniczne (*facet* oraz *receptacle*) dwa porty synchroniczne są zgodne jeśli obywda posiadają ten sam interfejs lub interfejs portu *receptacle* jest interfejsem bazowym interfejsu portu *facet* (innymi słowy połączenie zapewnia dostarczenie funkcjonalności wymaganej przez port *receptacle*),

porty asynchroniczne (*event sink, event emitter, event publisher*) dwa porty asynchroniczne są zgodne tylko wtedy gdy, jeden z nich produkuje, a drugi konsumuje zdarzenia tego samego typu[3].

Etap ten jest wykonywany, w razie pozytywnego zakończenia etapu 1.

Dodatkowo połączenie nie zostanie pozytywnie zwalidowane jeśli:

- podjęto próbę dołączenia portu *facet* do portu *receptacle*, który już posiada połączenie do innego portu synchronicznego i nie jest portem typu *multiple receptacle,*
- podjęto próbę dołączenia portu *event sink* do portu *envet emitter*, który już posiada połączenie do innego portu asynchronicznego.

Na samym początku walidacji sprawdzane jest również czy pomiędzy portami już nie utworzono połączenia by uniknąć nadmiarowości.

W przypadku łączenia portów komponentów, z których co najmniej jeden został już osadzony i uruchomiony może się zdarzyć, że źródło wiedzy na temat komponentów (*Interface Repository)*, nie posiada żadnych informacji na temat rozważanych typów portów - wówczas aplikacja poinformuje o tym fakcie (rysunek 4.30) a połączenie zostanie utworzone na żądanie i odpowiedzialność samego użytkownika.

Jeżeli użytkownik podejmie próbę połączenia portów których rodzaje lub typy na podstawie wiedzy zapisanej w *Interface Repository* nie są kompatybilne zostanie o tym poinformowany komunikatem w polu statusu edytora. Tabela 4.2 przedstawia poszczególne komunikaty informujące o niepowodzeniu walidacji.

[3] jest to wniosek empiryczny poparty testami w środowisku *OpenCCM*

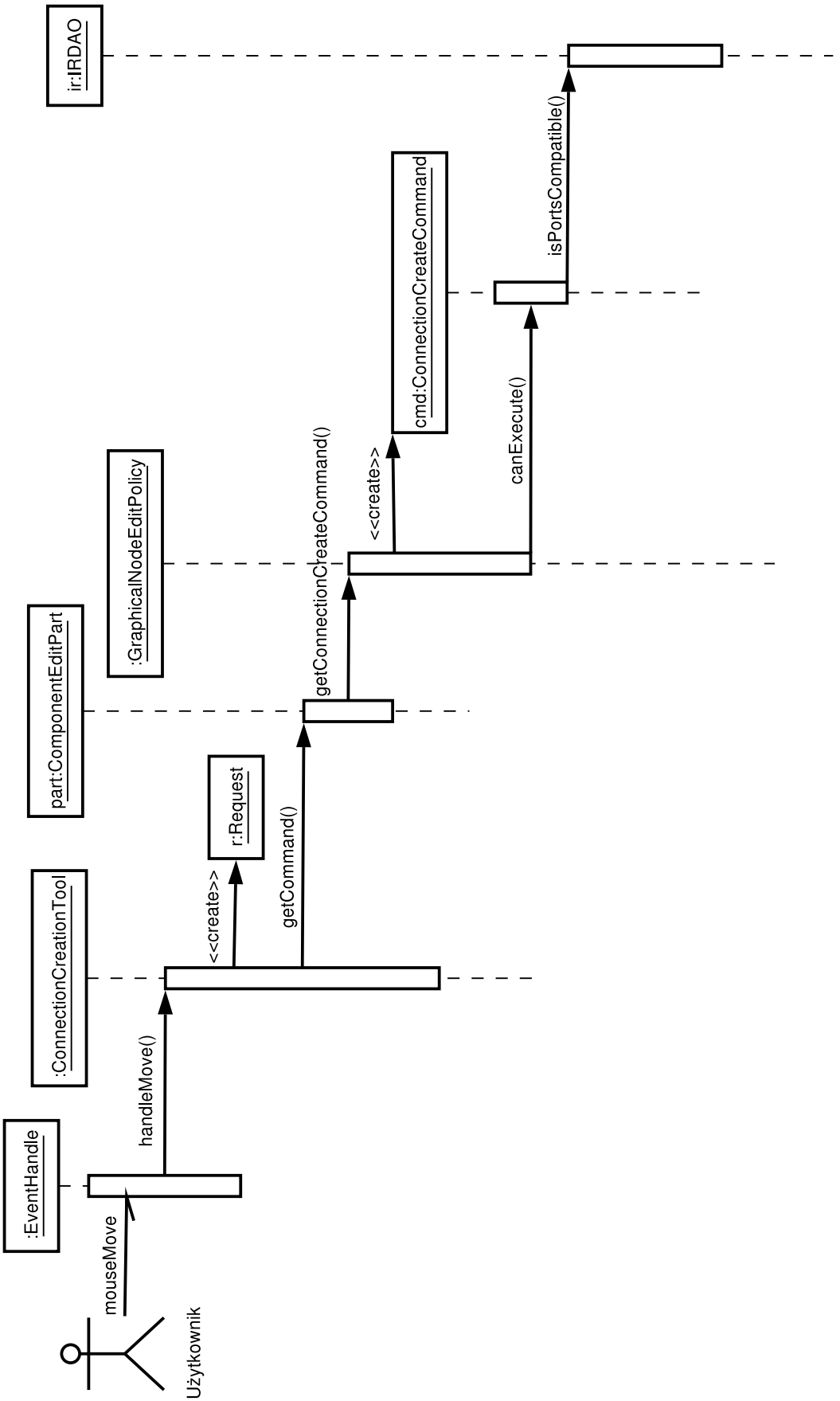
Komunikat	Przyczyna
Connection already exists!	Użytkownik próbuje połączyć porty pomiędzy, którymi istnieje juz połączenie
Facet can be connected only to Receptacle!	Użytkownik próbuje połączyć port *Facet* z portem innego rodzaju niż *Rceptacle*
Receptacle can be connected only to Facet!	Użytkownik próbuje połączyć port *Receptacle* z portem innego rodzaju niż *Facet*
Event Emitter can be connected only to Event Sink!	Użytkownik próbuje połączyć port *Event Emitter* z portem innego rodzaju niż *Event Sink*
Event Publisher can be connected only to Event Sink!	Użytkownik próbuje połączyć port *Event Publisher* z portem innego rodzaju niż *Event Sink*
Event Sink can be connected only to Event Emiter or Event Publisher!	Użytkownik próbuje połączyć port *Event Sink* z portem innego rodzaju niż *Event Emiter albo Event Publisher!*
Only multiple receptacle can be connected to many facets!	Użytkownik próbuje połączyć port *Receptacle* nie będący typu *multiple* z kolejnym portem *Facet*
Event Emitter can have only one connection!	Użytkownik próbuje połączyć port *Event Emiiter* z kolejnym portem *Event Sink*
Ports are not compatible!	Użytkownik próbuje połączyć porty których typy nie są zgodne na podstawie wiedzy zawartej w *Interface Repository*

Tablica 4.2. Komunikaty o braku zgodności łączonych portów.

Wymaganie to zostało zaimplementowane przy użyciu dedykowanej komendy środowiska *GEF*. Użyto opisywanej już w edytorze *Shapes Editor*, klasy *ConnectionCreateCommand*, w której zmodyfikowano metodę *canExecute,* tak by przeprowadzała walidację portów zgodnie z przedstawionym schematem. W tym celu odwołuje się do wiedzy zapisanej w *Interface Repository* poprzez klasę *IRDAO* .

Rysunek 4.31 przedstawia w jaki sposób akcja użytkownika, jaką jest tworzenie połączenia, zostaje zamieniona przez narzędzie środowiska *GEF (ConnectionCreationTool)* na żądanie, dla którego następnie jest generowana komenda (*ConnectionCreateCommand)*. Końcowy etap akcji prezentowany na diagramie sekwencji obrazuje współpracę stworzonej komendy z obiektem klasy IRDAO[4].

[4] w miejscu metody *isPortsCompatible()* w aplikacji wywoływana jest odpowiednia metoda z klasy *IRDAO*, zapewniająca walidację połączenia



Rysunek 4.31. Tworzenie komendy *ConnectionCreateComand* w odpowiedzi na akcje użytkownika

4.2.8. Generowanie pliku *.CAD*

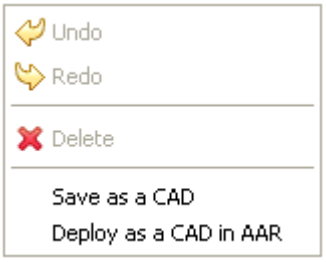
Główną funkcją aplikacji *CGE* jest generowanie pliku *.CAD* ze stworzonego modelu. Komponenty, ich własności, porty, miejsca alokacji znajdujące się na diagramie posiadają bezpośrednie odwzorowanie w budowanym deskryptorze. Generowany plik jest zgodny z definicją typu pliku *XML* znajdującą się na stronie:

http://openccm.objectweb.org/dtd/ccm/componentassembly.dtd.

Eksport diagramu do formatu pliku *.CAD* odbywa się poprzez wybranie z menu kontekstowego diagramu (prawy przycisk myszy na diagramie) odpowiedniej opcji. Istnieją dwa tryby zapisu pliku:

- ***Save as a CAD*** - polecenie to powoduje wyświetlenie okna dialagowego umożliwiającego określenie ścieżki, w której ma być zapisany generowany deskryptor.
- ***Deploy as a CAD in AAR*** - polecenie to powoduje wyświetlenie okna dialogowego umożliwiającego określenie lokalizacji pliku *.AAR,* do którego ma zostać wgrany generowany deskryptor.

Menu kontekstowe umożliwiające rozpoczęcie generowanie pliku *.CAD* przedstawia rysunek 4.32.



Rysunek 4.32. Menu kontekstowe generowania deskryptora *.CAD*

W obu trybach zapisu zapamiętywana jest ostatnio wybrana przez użytkownika ścieżka, co ma na celu usprawnienie pracy z edytorem. Wiąże się to z tym, że zwykle użytkowik operuje na plikach znajdujących się w tej samej lokalizacji.

Klasą tworzącą menu kontekstowe jest klasa *CADEditorContextMenuProvider.* Z tej klasy wywoływane mogą być pewne akcje (obiekty klas rozszerzających klasę *Action*). Dla nas interesujące są dwie z nich:

CADPlainExporter - akcja umożliwiająca zapis pliku *.CAD* do wybranej lokalizacji (opcja *Save as a CAD*),

DeployAction - akcja umożliwiająca wgranie pliku *.CAD* do wybranego archiwum *.AAR* (opcja *Deploy as a CAD in AAR*).

Generowane dane mają mieć postać pliku *XML*. Tworzenie danych w takiej postaci ułatwia język przekształceń *XSLT* (ang. *Extensible Stylesheet Language Transformations*, przekształcenia rozszerzalnego języka arkuszy stylów), który jest językiem przekształceń, pozwalającym na określanie sposobów przekształcania danych *XML* na

inne formaty. Na przykład *XSLT* można zastosować do wygenerowania na podstawie danych *XML* pliku *HTML* lub pliku *XML* o innej strukturze. *XSLT* można także wykorzystać do wygenerowania zwyczajnego pliku tekstowego lub do umieszczenia informacji w dokumencie zapisanym w zupełnie innym formacie. Można także wykorzystać *XSLT* do przekształcania danych, które nie są zapisane w formacie *XML*. Właśnie z tej funkcjonalności korzystaliśmy w naszym projekcie. Funkcjonalność generowania deskryptora asemblacji z wykorzystaniem języka przekształceń *XSLT* implementują następujące klasy z pakietu *pl.go.die.editor.xml*:

DataGenerator - klasa generująca poszczególne znaczniki z wykorzystaniem języka przekształceń *XSLT,*

Markup - abstrakcyjna klasa zapewniająca podstawową funkcjonalność dla rozszerzających ją klas,

PrecedentMarkup - klasa rozszerzająca klasę *Markup;* implementuje funkcjonalność znacznika mogącego posiadać znaczniki potomne,

LastMarkup - klasa rozszerzająca klasę *Markup;* implementuje funkcjonalność znacznika nie mogącego posiadać znaczników potomnych,

Properties - klasa zawierająca statyczne pola identyfikujące poszczególne sekcje w generowanym deskryptorze.

Zaprezentowane rozwiązanie prezentuje dużą elastyczność i spore możliwości w tworzeniu dowolnych plików o strukturze *XML*.

4.2.9. Odczyt i zapis diagramu z/do pliku

Edytor *CGE* pozwala na zapis i odczyt edytowanego diagramu. Zapisowi podlegają:

- komponenty (z wyjątkiem komponentów osadzonych), tj. rozmiar komponentu, jego położenie, porty oraz przypisanie do logicznego i fizycznego miejsca alokacji,
- połączenia pomiędzy portami (z wyjątkiem połączeń związanych z portami komponentów już osadzonych),
- logiczne miejsce alokacji,
- odwzorowanie logicznych miejsc na fizyczne miejsca alokacji,
- parametry połączenia z *Name Service i Interface Repository,*
- wybrany widok modelu (komponentów, logicznych miejsc alokacji, fizycznych miejsc alokacji),
- wybrany stopień przybliżenia (*zoom*).

W przypadku odczytu zapisanego diagramu w momencie gdy użytkownik nawiąże połączenie z *Name Service* (dokładna procedura została opisana w 4.2.5) odwzorowanie jest uaktualniane. Zarówno komponenty umieszczone już w środowisku jak i połączenia z nimi związane nie podlegają zapisowi ze względu na ich ulotność, tj. informacja z nimi związana uległaby zapewne i tak dezaktualizacji w momencie ponownego wczytania, przy założeniu pracy z dynamicznym środowiskiem.

Odczyt i zapis diagramu został zrealizowany dzięki standardowej metodzie serializacji obiektów jaką dostarcza język Java. Każdy element modelu implementuje interfejs *java.io.Serializable.* Klasa edytora *CADEditor* w metodzie *doSave* zapisuje obiekt diagramu, a w metodzie *setInput* tworzy instancje diagramu na podstawie informacji

zapisanej w pliku. Środowisko *GEF* wymaga użycia stosu poleceń, aby edytor poprawnie monitorował i zapisywał zmiany w edytowanym diagramie. Stos poleceń zostanie opisany w 4.2.12.

4.2.10. Możliwość jednoczesnej pracy na wielu diagramach

Edytory korzystające z biblioteki *GEF* posiadają własność wielodiagramowości. Własność ta pozwala na otwarcie wielu diagramów i jednoczesnej pracy na nich. Dzięki zastostowaniu osobnej instancji palety dla każdego z otwartych diagramów możliwe jest posiadanie otwartych diagramów z wczytanymi danymi (komponentami, miejscami alokacji) pochodzącymi z różnych lokalizacji. Właściwości diagramu tj. stopień przybliżenia, wybrany widok, są również pamiętane osobno dla każdego z diagramu, co powoduje przewidywalne zachowanie systemu podczas przełączania pomiędzy diagramami.

Jednoczesne korzystanie z wielu diagramów umożliwia porównywanie tworzonego modelu z innymi już istniejącymi oraz tworzenie różnych alternatywnych rozwiązań jednocześnie. Rozwiązanie to osiągnęliśmy dzięki dostosowaniu funkcjonalności klas:

CADEditor - klasa reprezentująca edytor,
CCMDiagram - klasa reprezentująca diagram edytora; posiada statyczną metodę *getCCMDiagram* zwracającą obiekt z puli otwartych diagramów.

4.2.11. Dodawanie elementów do diagramu za pomocą palety

Edytor *CGE* posiada paletę zawierającą następujące elementy (przedstawione na rysunku 4.33):

- narzędzia do zaznaczania (opisane w 4.2.13),
- logiczne miejsca rozmieszczenia (opis w 4.2.2),
- fizyczne miejsca rozmieszczenia (osobno pogrupowane są serwery komponentów i węzły - opis w 2.2.3),
- komponenty (definicje wczytane z *Interface Repository* - opis w 4.2.5),
- osadzone komponenty (opis w 4.2.6).

Ze względu na sposób użycia wymienione elementy można podzielić na dwie grupy:

- elementy dodawane do diagramu za pomocą mechanizmu *przeciągnij i upuść (drag & drop)* - należą tu obydwa rodzaje komponentów,
- elementy używane w trybie *znacznika (markera),* - użytkownik po wybraniu elementu z palety zaznacza komponenty na diagramie, co skutkuje wykonaniem określonej akcji edycyjnej modelu - należą tu obydwa rodzaje miejsc alokacji oraz narzędzia do zaznaczania.

Rysunek 4.33. Opis palety edytora *CGE*

Tryby pracy palety

Zawartość palety zmienia się w zależności od trybu w jakim znajduje się edytor *CGE*. Tryby przedstawia tabela 4.3.

Uaktualnianie zawartości palety

Zawartość palety w edytorze ulega zmianie w odpowiedzi na zmiany zachodzące w środowisku. Edytor *CGE* wykrywa i podejmuje odpowiednie akcje w celu aktualizacji palety w następujących sytuacjach:

- zniknięcie/pojawienie się nowego fizycznego miejsca alokacji,
- dodanie/usunięcie logicznego miejsca alokacji,
- zniknięcie/pojawienie się nowego rozmieszczonego komponentu,
- zmiana zawartości *Interface Repository*.

Użytkownik zostaje poinformowany o zmianie środowiska za pomocą odpowiedniego komunikatu (ostrzeżenia zostały omówione wcześniej, przy okazji opisu poszczególnych elementów znajdujących się na palecie).

Tryb	Zawartość palety
nowy plik	narzędzia do zaznaczania, logiczne miejsca alokacji
bezpołączeniowy	narzędzia do zaznaczania, logiczne miejsca alokacji, fizyczne miejsca alokacji zapisane w pliku
połączony z Naming Service	narzędzia do zaznaczania, logiczne miejsca alokacji, aktualne fizyczne miejsca alokacji oraz rozmieszczone komponenty
połączony z Interface Repository	narzędzia do zaznaczania, narzędzie do łączenia portów, komponenty
połączony z serwisami CCM	narzędzia do zaznaczania, narzędzie do łączenia portów, komponenty, aktualne fizyczne miejsca alokacji oraz rozmieszczone komponenty

Tablica 4.3. Zawartość palety w różnych trybach pracy edytora

Opisana funkcjonalność została zrealizowana przy pomocy zestawu następujących klas:

CADComponentFactory klasa *fabryka*[5] komponentów, których definicje wczytano uprzednio z *Interface Repository.*

DeployedComponentFactory klasa *fabryka* komponentów już osadzonych w środowisku, wykrytych w *Name Service.*

CADEditorPaletteFactory klasa tworząca i zarządzająca paletą, pełni rolę obserwatora[6] zdarzeń generowanych przez *DestinationRepository* w wyniku zmiany zestawu logicznych lub fizycznych celów alokacji.

ILogicalDestinationListListener, IPhysicalDestinationListener interfejsy implementowane przez *CADEditorPaletteFactory,* dzięki którym fabryka ta informowana jest o zmianach w zestawie miejsc alokacji.

pakiet pl.go.die.editor.tools zawiera zestaw klas implementujących narzędzia do zaznaczania elementów diagramu, m.in *LogicalDestinationMarker, PhysicalDestinationMarker,* a także *ComponentsSelectionTool* i *PortsSelectionTool;* oprócz narzędzi zdefinowane są w tym pakiecie reprezentujące je kontrolki dodawane do palety (np. *LogicalDestinationMarkerEntry*).

pakiet pl.go.die.editor.views.mapping zawiera implementacje widoku *Destinations Mapping View,* pozwalającego na zmianę odwzorowania logicznych miejsc na fizyczne miejsca alokacji, również pełni rolę obserwatora dla zdarzeń generowanych przez *DestinationRepository.*

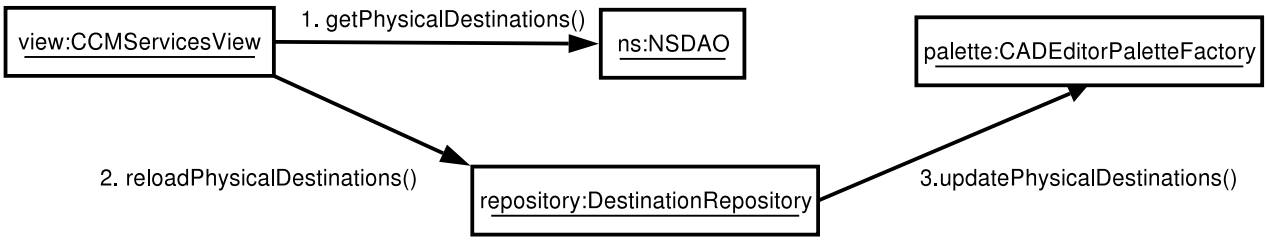
CCMServicesView opisany w poprzednich punktach.

IRDAO, NSDAO, DestinationRepository również zostały już omówione.

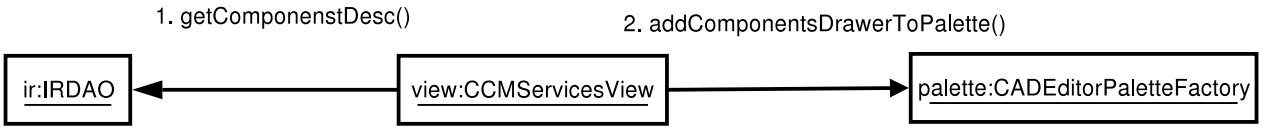
Przykłady współpracy pomiędzy klasami przedstawiają diagramy kolaboracji (rysunek 4.34, rysunek 4.35, rysunek 4.36).

[5] realizacja wzorca projektowego zwanego *abstract factory*

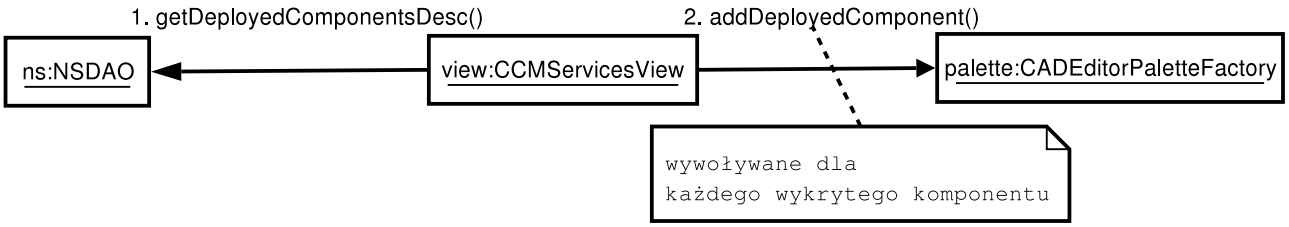
[6] realizacja wzorca projektowego zwanego *observer*



Rysunek 4.34. Wczytanie fizycznych miejsc alokacji



Rysunek 4.35. Wczytanie definicji komponentów



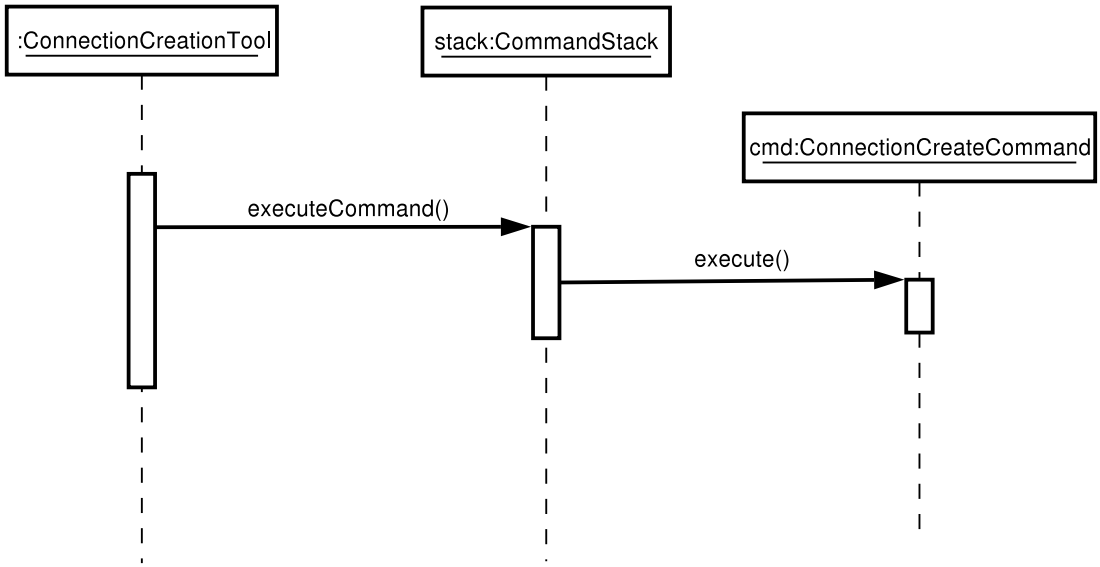
Rysunek 4.36. Wykrycie osadzonych komponentów

4.2.12. Obsługa stosu poleceń

Edytor *CGE* w pełni wykorzystuje dostarczany wraz ze środowiskiem *GEF stos poleceń (command stack)*. Modyfikacje modelu wykonywane są poprzez komendy, a te z kolei uruchamiane sa na stosie poleceń. Komenda zawiera metodę *canUndo* wskazującą czy może zostać cofnięta oraz metody *redo* i *undo* określające co ma się stać przy zdjęciu komendy ze stosu, a co przy ponownym odłożeniu. Komendy według tego kryterium można podzielić na dwie grupy:

- wspierające mechanizm *redo/undo,*
- niepodlegające wycofaniu.

Wykorzystanie stosu poleceń nie tylko wzbogaca edytor o dodatkową funkcjonalność, do której przywykł użytkownik. Każde odłożenie i wykonanie na stosie komendy informuje edytor o zmianie modelu i co za tym idzie o potrzebie jego zapisu. Z tego względu wszystkie modyfikacje modelu jakie należy trwale zachować edytor *CGE* przeprowadza za pomocą komend. Rysunek4.37 przedstawia diagram sekwencji wykonania komendy *ConnectionCreateCommand* przy użyciu stosu poleceń.



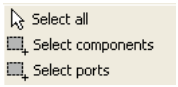
Rysunek 4.37. Wykonanie komendy *ConnectionCreateCommand*

4.2.13. Zróżnicowane tryby zaznaczania elementów

Aby ułatwić pracę w edytorze *CGE* wprowadzono trzy metody selekcji elementów poprzez obrysowanie lub kliknięcie:

1. **Zaznaczenie każdego elementu (*Select all*)**
   Tryb użyteczny, gdy zachodzi potrzeba przemieszczenia licznej grupy elementów diagramu - został zaimplementowany przy użyciu standardowego narzędzia w ramach środowiska *GEF* zwanego *PanningSelectionTool*.
2. **Zaznaczenie wyłącznie komponentów (*Select components*)**
   Tryb wykorzystywany w sytuacji edycji wspólnych właściwości grupy komponentów - został zrealizowany przy użyciu klasy *ComponentsSelectionTool*.
3. **Zaznaczenie wyłącznie portów (*Select ports*)**
   Tryb używany w przypadku edycji wspólnych właściwości portów - został zaimplementowany przy użyciu klasy *PortsSelectionTool*.

Rysunek 4.38 prezentuje ikony trybów zaznaczanie w edytorze.



Rysunek 4.38. Tryby zaznaczania

Klasą bazową dla *ComponentsSelectionTool i PortsSelectionTool* stanowi klasa *CADMarqueeSelectionTool* zawierająca metodę *isRequiredToSelect(EditPart part)*. Klasy dziedziczące przesłaniając tę metodę określają kryteria jakie musi spełnić element diagramu, by zostać zaznaczony w danym trybie.

4.2.14. Drukowanie diagramu

Wprowadziliśmy możliwość drukowania stworzonego diagramu. Druk możliwy jest na 3 sposoby:

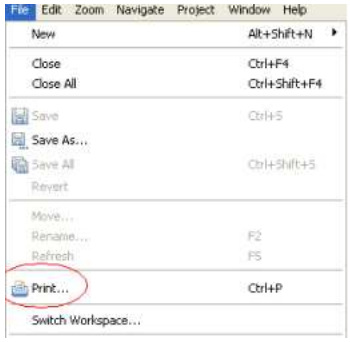
- Kliknięcie ikonki drukarki w pasku górnym środowiska *Eclipse* (rysunek 4.39).

Rysunek 4.39. Ikona drukowania

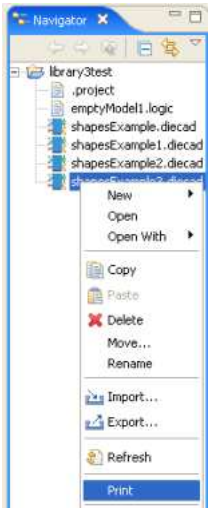
- Wybór ikonki drukarki z menu *File* lub kombinacja klawiszy *CTRL-p* (rysunek *4.40*).
- Wybór z menu kontekstowego w panelu *Navigator* środowiska *Eclipse* funkcji *Print.* W tej opcji pojawia się możliwość wyboru 4 widoków drukowania: *tile, fit page, fit height, fit width*, w zależności od potrzeb użytkownika. Ta opcja umożliwia wydruk dowolnego (nawet nieotwartego) diagramu znajdującego się w panelu *Navigator* (rysunek 4.41).

Na potrzeby obsługi drukowania diagramu powstały następujące klasy znajdujące się w pakiecie *pl.go.die.editor.actionbuttons*:

CADPrintAction - klasa obsługująca akcję wydruku.
PrintModeDialog - okno dialogowe wyświetlające się po wybraniu opcji drukowania. Pozwala na wybór trybu wydruku.



Rysunek 4.40. Drukowanie z menu



Rysunek 4.41. Drukowanie z panela *Navigator*

4.3. Podsumowanie

Implementację tworzono w oparciu o model ewolucyjny rozwoju oprogramowania, z jednym wyjątkiem: prace nad wymaganiem 4.2.2 zakończyły się porzuceniem prototypu przedstawiającego miejsca alokacji jako kontenery. Należy podkreślić, że dzięki zastosowaniu środowiska *GEF* zminimalizowano liczbę zależności pomiędzy implementacjami poszczególnych wymagań, czemu służy stosowany wzorzec architektoniczy *Model - Widok - Kontroler*. Zrealizowano obydwa główne cele aplikacji, tj. stworzenie graficznego interfejsu do budowy deskryptora *CAD* oraz walidację semantyczną połączeń między portami.

Rozdział 5

Asemblacja aplikacji komponentowych za pomocą edytora CGE

Celem rozdziału jest przedstawienie czytelnikowi pełnej listy kroków jaką należy wykonać, by w edytorze *CGE* stworzyć deskryptor CAD aplikacji komponentowej. Jako przykład aplikacji posłuży implementacja rozwiązania *problemu jedzących filozofów* (opisany w 2.1.4) dostarczana wraz ze środowiskiem *OpenCCM.* Aplikacja składa się z trzech rodzaji komponentów:

Philosopher modeluje filozofa posiada porty:
- *left* - *receptacle* typu *Fork,*
- *right* - *receptacle* typu facet *Fork,*
- *info* - *event publisher* generujący zdarzenia typu *StatusInfo.*

ForkManager posiada port *facet* udostępniający interfejs *Fork.*
Observer posiada port *event sink* konsumujący zdarzenia typu *StatusInfo.*

Stworzenie aplikacji stanowiącej symulacje problemu jedzących filozofów w modelu CCM polega na połączeniu kompatybilnych portów opisanych komponentów.

Implementacja poszczególnych komponentów wraz ze skryptem kompilującym źródła i tworzącym archiwum aplikacji (plik **.aar*) zamieszczamy na płycie CD. Dalszy opis zakłada, że czytelnik posiada już wygenerowany plik **.aar* oraz i skupia się na złożeniu aplikacji w określonej konfiguracji:

- liczba filozofów: pięć,
- liczba widelców: pięć,
- jeden obserwator.

5.1. Krok I: Skonfigurowanie środowiska uruchomieniowego OpenCCM

Zanim przystąpimy do tworzenia diagramu niezbędne jest skonfigurowanie środowiska *OpenCCM* zgodnie z następującą listą poleceń:

1. Instalacja repozytorium konfiguracji środowiska *OpenCCM.*

   ```
   ccm_install
   ```

2. Uruchomienie *Interface Repository.*

   ```
   ir3_start
   ```

3. Uruchomienie *Name Service.*

   ```
   ns_start
   ```

4. Uruchomienie serwerów komponentów.

   ```
   jcs_start ComponentServer1
   jcs_start ComponentServer2
   ```

5. Uruchomienie *DCI Manager.*
   OpenCCM Distributed Computing Infrastructure (DCI) tworzy zbiór rozproszonych węzłów. Manager zarządza domeną DCI, rejestruje węzły.

   ```
   dci_start  DefaultDCI
   ```

6. Uruchomienie *Assembly Factory Manager.*
   OpenCCM Assembly Factory Manager służy do tworzenia, budowania, niszczenia i zarządzania asemblacją.

   ```
   factory_start DefaultFactory
   ```

7. Uruchomienie węzła.
   OpenCCM Node Manager reprezentuje logicznie węzły DCI. Pozwala na uruchomienie i zatrzymanie serwerów komponentów.

   ```
   node_start MainNode
   ```

5.2. Krok II: Nawiązanie połączenia z serwisami OpenCCM

Proces opisany w poprzednum podpunkcie można przeprowadzić lokalnie na tym samym komputerze, na którym będziemy asemblować aplikację. Uruchamiamy edytor CGE i wykonujemy następujące kroki:

1. **Tworzymy nowy diagram.**
   Z menu *File* klikamy *Other* i wybieramy jedyny kreator z kategorii *OpenCCM descriptors editors.* Podajemy nazwę tworzonego diagramu.

2. **Łączenie z *Interface Repository.***
   W widoku *CCM Services View* (jeśli nie otwarty został do tej porty można go uaktywnić w menu *Windows -> Show View -> Other*) wybieramy zakładkę *Interface Repository.* Wpisujemy adres *IOR Interface Repository* (np. *localhost,* jeśli mamy skonfigurowane lokalnie repozytorium albo zdalne, którego identyfikator dostępny jest pod adresem *http*). Na koniec klikamy przycisk *(Re)connect.* Otrzymamy listę komponentów na palecie edytora.
3. **Łączenie z *Name Service*** (opcjonalne na tym etapie).
   W widoku *CCM Services View* wybieramy zakładkę *Name Service.* Podajemy adres (*localhost*, adres *http* albo adres *corbaloc*) i naciskamy przycisk *(Re)connect.* Krok ten można równie dobrze wykonać bezpośrednio przed rozpoczęciem odworowywania logicznych miejsc alokacji na fizyczne. Otrzymamy listę fizycznych miejsc alokacji na palecie edytora.

5.3. Krok III: Dodanie komponentów do diagramu i ich konfiguracja

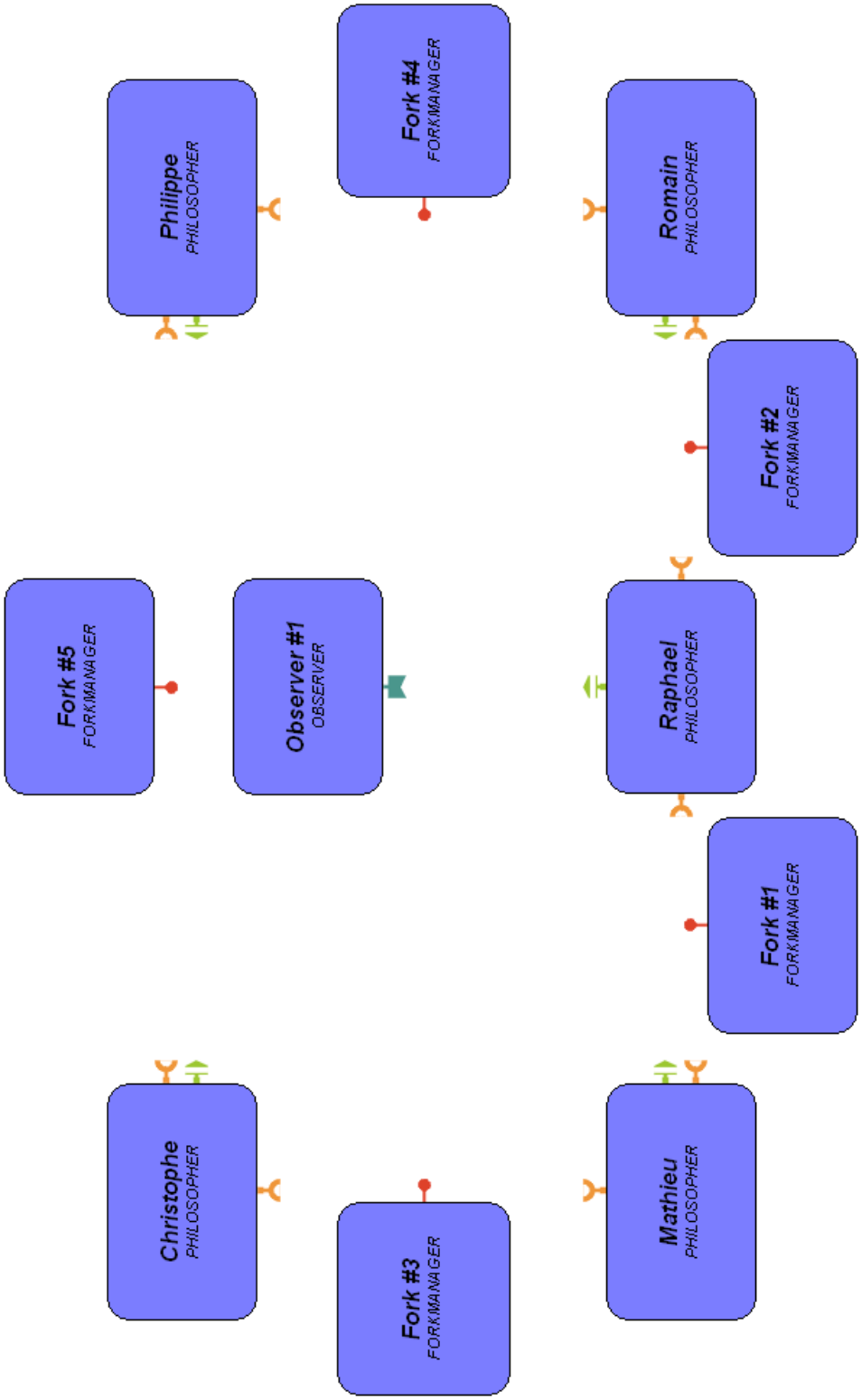
Na tym etapie zaczynamy właściwy proces asemblacji aplikacji. Przeciągamy z palety na diagram po jednym z każdego typu komponentów:

- *Observer,*
- *ForkManager,*
- *Philosopher.*

W widoku *Properties* (jeśli nie otwarty do tej pory to możemy go uaktywnić z menu *Windows -> Show View*) edytujemy właściwości komponentu. Edytor *CGE* został tak zrealizowany, że domyślnie wartości właściwości odpowiadają konwencji stosowanej w środowisku *OpenCCM,* dlatego konfiguracja powinna się ograniczyć jedynie do paru czynności:

- wymazanie zawartości właściwości *Component CPF File* dla komponentów *Observer* oraz *ForkManager*,
- wprowadzeniu właściwości *Component CPF File* dla każdego komponentu *Philosopher* - jedna z następujących:
  — META-INF/philippe.cpf,
  — META-INF/mathieu.cpf,
  — META-INF/christophe.cpf,
  — META-INF/romain.cpf,
  — META-INF/raphael.cpf.
- Następnie dodajemy po cztery instancje komponentu *ForkManager* oraz *Philosopher* - warto zauważyć, że raz zmienione właściwości danego typu komponentu stają sie dla niego wartościami standardowymi. Co za tym idzie dla kolejnych instancji typu komponentu *Philosopher* należy zmienić właściwość *Component CPF File* (tak byśmy mieli skonfigurowanych różnych filozofów), a dla instancji typu komponentu *ForkManager* nie trzeba nic zmieniać.

Po tym etapie powinniśmy otrzymać analogiczny diagram do tego przedstawionego na rysunku 5.1 (rozmieszczenie komponentów jest dowolne).



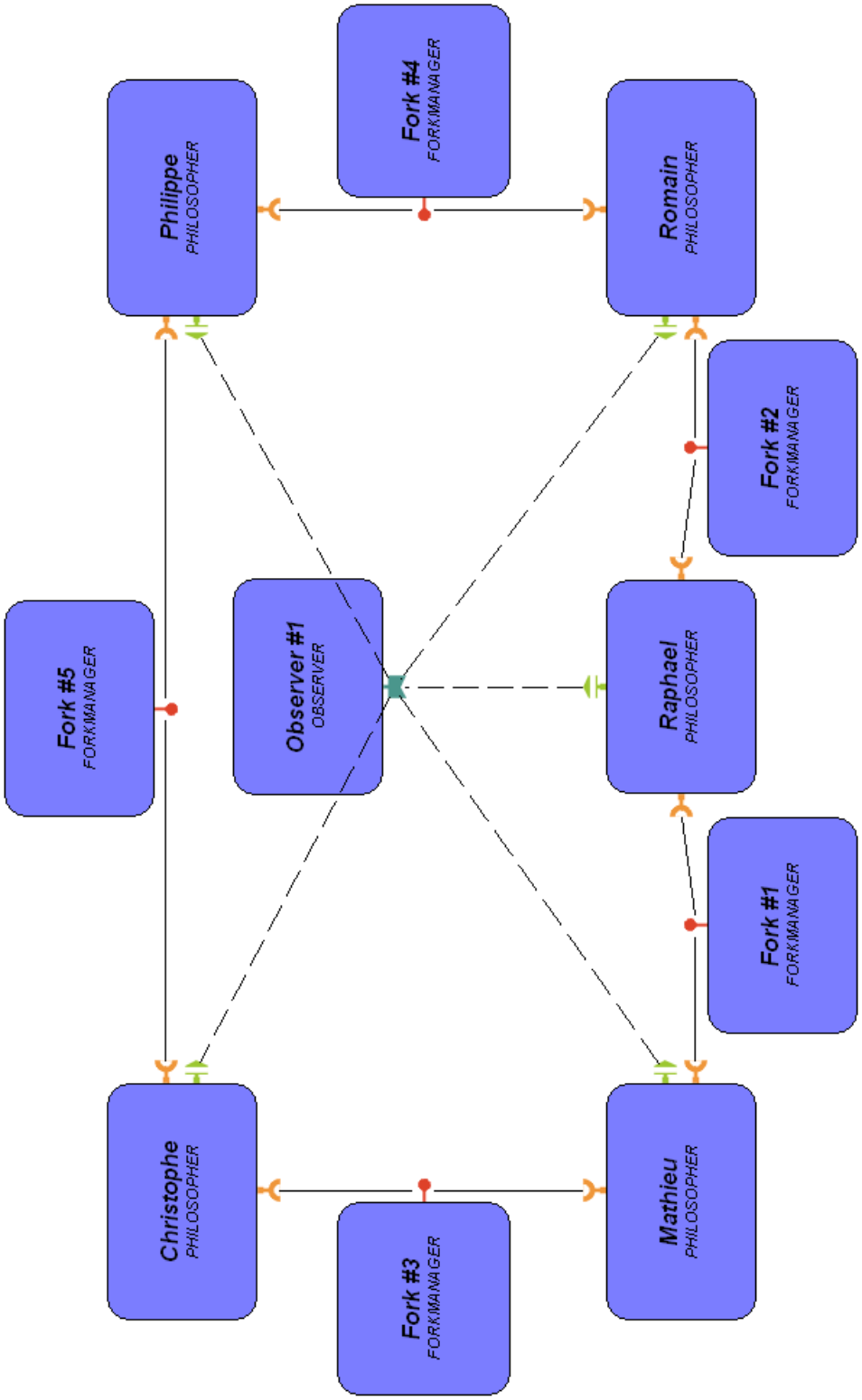
Rysunek 5.1. Komponenty używane w asemblacji

5.4. Krok IV: Połączenie portów komponentów

Łączymy porty według następującej zasady:

- każdy port *event publisher* komponentu *Philosopher* łączymy z portem *event sink* komponentu *Observer*,
- każdy port *facet* komponentu *ForkManager łaczymy z* dwoma portami *receptacle* należącymi do różnych instancji komponentu *Philosopher.*

Efekt powinien być podobny do przedstawionego na rysunku 5.2.



Rysunek 5.2. Połączenia między komponentami

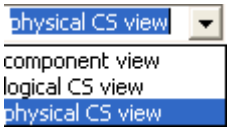
5.5. Krok V: Rozmieszczenie komponentów w środowisku

Przyporządkowanie komponentów do logicznych miejsc alokacji

Mamy trzy typy komponentów, niech każdy typ komponentu zostanie osadzony w innym logicznym miejscu alokacji, np.:

- *Observer* w *domyślnym* miejscu osadzenia,
- *Philosopher w LDS1,*
- *ForkManager w LDS2.*

Przyporządkowania komponentów do logicznych miejsc alokacji dokonujemy klikając najpierw na palecie ikonę żądanego logicznego miejsca alokacji, a następnie instancje komponentów, które mają być w nim osadzone. Proces ten najlepiej dokonać po przełączeniu na widok logiczny używając do tego kontrolki pokazanej na rysunku 5.3.



Rysunek 5.3. Przełącznik widoków

Efekt powinien być podobny do przedstawionego na rysunku5.4.

Przypisanie fizcznych miejsc alokacji do logicznych miejsc alokacji

Uwaga: Jeżeli edytor jeszcze nie nawiązał połączenia z Name Service należy teraz to wykonać zgodnie z opisem w 5.2.

Odwzorowanie najprościej wykonać z poziomu widoku *Destinations Mapping View* (dostępnego w menu *Window->Show View -> Other*, kategoria *CAD Graphical Editor*). Ponieważ mamy wystarczającą liczbę fizycznych miejsc alokacji możemy przypisać po jednym fizycznym miejscu dla każdego logicznego miejsca (nic nie stoi na przeszkodzie by spróbować innych odwzorowań). Przyjęte przypisanie przedstawia rysunek5.5.

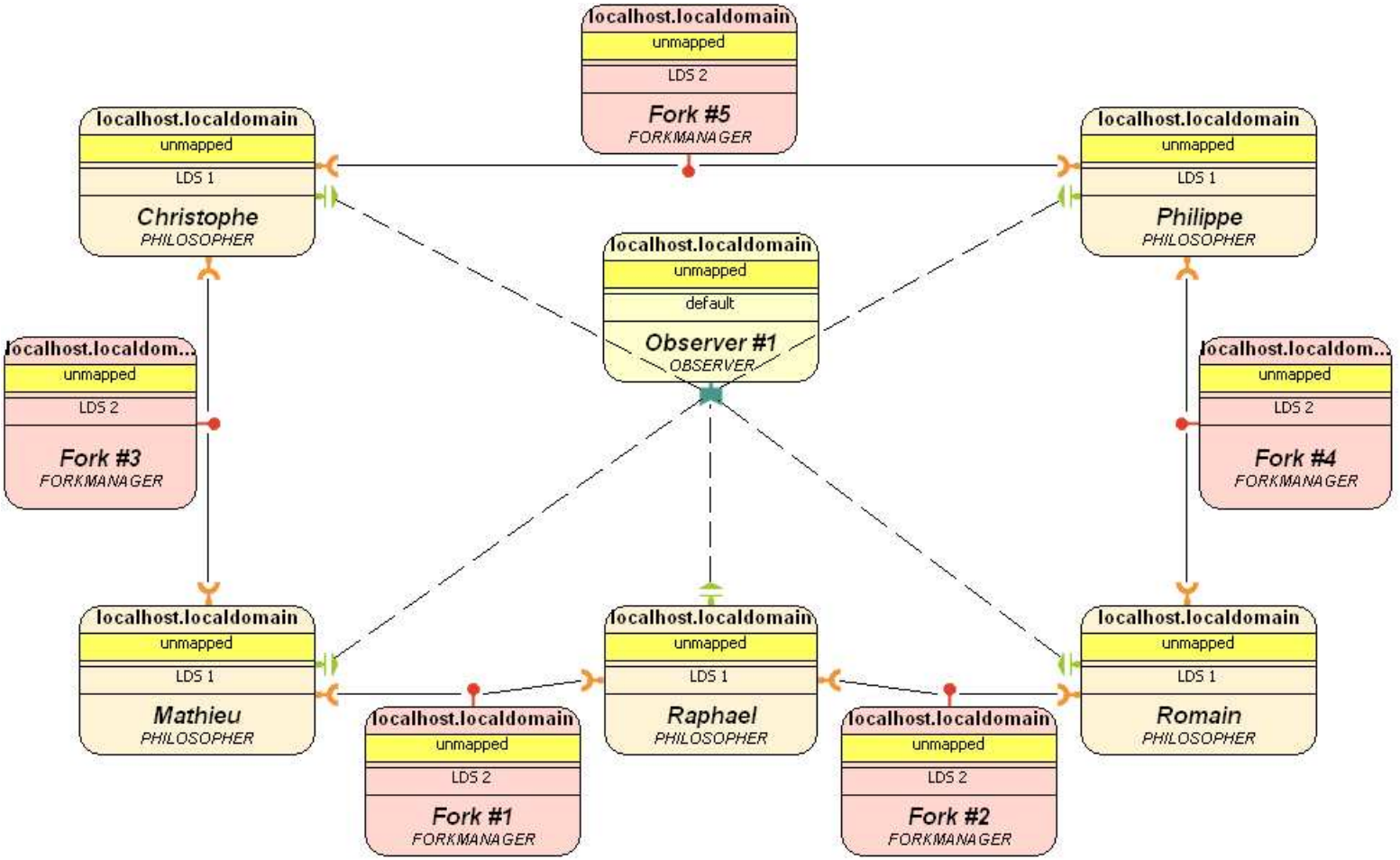
Uruchomienie aplikacji komponentowej

Na koniec generujemy deskryptor *CAD* i umieszczamy w archiwum aplikacji (plik **.aar*) za pomocą akcji *Deploy as a CAD in AAR*, dostępnej z menu kontekstowego edytora *CGE.*

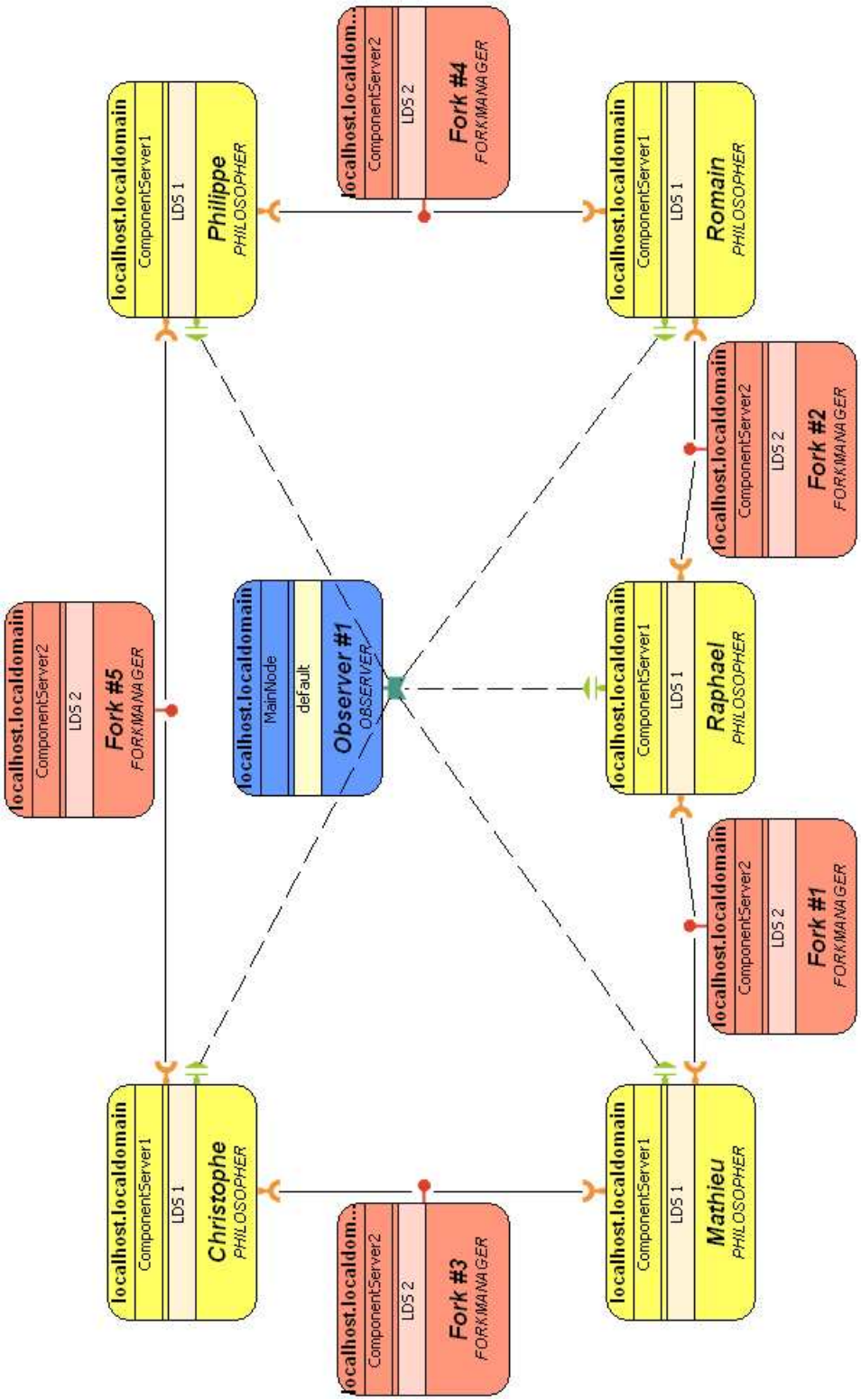
Tak przygotowaną aplikację uruchamiamy za pomocą polecenia:

```
ccm_deploy DininigPhilosophers.aar
```

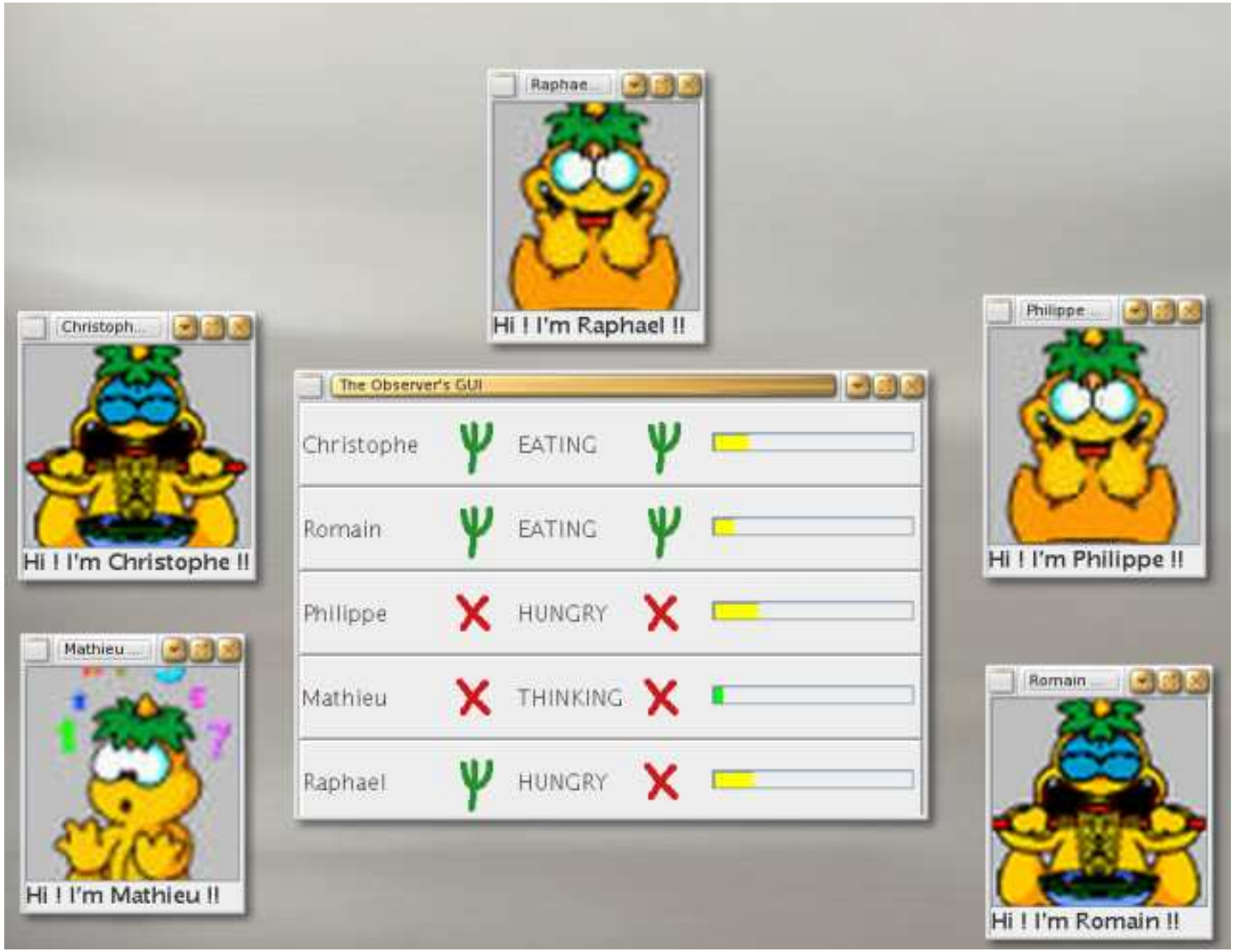
Wygląd osadzonych i uruchomionych komponentów współpracujących w ramach jednej asemblacji opisanej stworzonym deskryptorem *CAD* prezentuje rysunek5.6.



Rysunek 5.4. Przypisanie komponentów do logicznych miejsc alokacji (*unmapped* - wariant brak połączenia do *Name Service* lub brak dostępnych fizycznych miejsc alokacji komponentów - jeżeli edytor nawiązał połączenie do serwisu to w miejscu *unmapped* będzie domyślne fizyczne miejsce alokacji)



Rysunek 5.5. Tworzony diagram w widoku fizyczynych miejsc osadzenia



Rysunek 5.6. Uruchomiona aplikacja pięciu filozofów

5.6. Podsumowanie

Edytor *CGE* przyspiesza znacząco proces tworzenia aplikacji komponentowej na etapie asemblacji. W bardzo łatwy sposób można wygenerować różne wersje aplikacji opartej o te same komponenty. Użytkownik, w pełni korzystając ze wsparcia edytora dla walidacji połączeń, otrzymuje jako produkt poprawny zarówno syntaktycznie jak i semantycznie deskryptor *CAD*.

Obsługa aplikacji nie sprawia większych problemów, co zostało udowodnione poprzez przeprowadzenie z powodzeniem zajęć dydaktycznych na Katedrze Informatyki. Podczas tych zajęć studenci zapoznali się w praktyce z modelem *CCM*, a edytor *CGE* wykorzystali jako narzędzie pomocnicze do przeprowadzania etapu asemblacji aplikacji rozproszonej.

Rozdział 6

Zakończenie

Po zapoznaniu się z *CORBA Component Model* jesteśmy przekonani o jego dużej funkcjonalności, ale nie sposób przemilczeć duży narzut pracy towarzyszący tworzeniu nawet najprostszej aplikacji w oparciu o środowisko *CCM*. Skonstruowany edytor *CGE* ma za zadanie ułatwić jeden z etapów budowy oprogramowania, jakim jest proces asemblacji. Faza ta bez pomocy aplikacji wspomagających jej przeprowadzenie jest uciążliwa, co uzasadnia potrzebę stworzenia narzędzia automatyzującego ten proces.

Powstały edytor *CGE* cechuje się bogatą funkcjonalnością przede wszystkim dzięki wykorzystaniu biblioteki *GEF*. Użycie jej umożliwiło szybki rozwój aplikacji, i co najważniejsze, mogliśmy się od razu skupić na implementacji funkcjonalności związanej z modelem *CCM* - podstawowe wymagania wspólne dla wszystkich graficznych edytorów zostały zapewnione przez bibliotekę *GEF*. Należy nadmienić, że dokumentacja biblioteki *GEF* nie jest zbyt obszerna, ale za to dostępne są liczne przykłady użycia co powoduje, że jej wykorzystanie najlepiej zacząć od rozwijania i przystosowywania do własnych potrzeb istniejących, przykładowych edytorów - taką też drogę wybraliśmy budując aplikację *CGE* w oparciu o omówiony edytor *Shapes Editor*. Po dogłębnym rozpoznaniu biblioteki *GEF* jesteśmy skłonni przyznać, że jej twórcy stworzyli stabilne środowisko o szerokim spektrum zastosowań. Na podstawie własnych doświadczeń (zmiana sposobu przydziału komponentów do miejsc alokacji) przekonaliśmy się, że użyty w niej wzorzec architektoniczny *Model - Widok - Kontroler* gwarantuje dużą elastyczność i pozwala w łatwy sposób dostosować rozwijany produkt do zmieniających się wymagań użytkownika.

Biblioteka *GEF* stanowi wtyczkę do środowiska *Eclipse,* z pomocą której w łatwy sposób mogliśmy się zintegrować z cieszącym się dużą popularnością wśród informatyków narzędziem. Tworząc edytor *CGE* jako wtyczkę do środowiska *Eclipse,* zyskaliśmy możliwość wykorzystania w pełni zawansowanych funkcji takich jak, np. widoki, zakładki, eksplorator plików, stos poleceń, rozbudowane paski narzędzi i menu kontekstowe.

Z perspektywy czasu za kluczowy krok w budowie aplikacji postrzegamy dobór składowych komponentów. Sprawdzone i stabilne komponenty pozwoliły nam na pełną realizacje postawionych przed aplikacją wymagań. Jednocześnie upewniliśmy się w przekonaniu, że programowanie komponentowe stanowi dalszy krok w procesie automatyzacji tworzenia oprogramowania. Póki co, pełną automatyzację uniemożliwiają występujące braki zgodności między interfejsami poszczególnych komponentów - i tu właśnie z pomocą przychodzi model *CCM* z klarownym opisem funkcji realizowanych przez jedne komponenty a wymaganych przez inne.

Sam model na nie wiele się zda, dopóki nie doczeka się stabilnej i łatwej w użyciu implementacji. Naszym zdaniem środowisko *OpenCCM,* póki co, takim rozwiązaniem nie jest z powodu braku programów wspomagających skomplikowany proces tworzenia aplikacji rozproszonych. Należy jednak docenić wsparcie zapewniane przez *OpenCCM* na etapie uruchomienia aplikacji rozproszonej (zestaw skryptów konfiguracyjnych, serwisy, serwery, węzły i całe rozproszone środowisko). Remedium na opisane słabości implementacji byłoby stworzenie zintegrowanego zbioru wtyczek do środowiska *Eclipse,* które automatyzowałyby tworzenie poszczególnych deskryptorów. Zadanie to wydaje się tym bardziej osiągalne, że ważny element tego zestawu, edytor *CGE,* powstał i może służyć jako dobry przykład dla budowy pozostałych.

Nasza aplikacja została zaprezentowana studentom i wykorzystywana przez nich na zajęciach z przedmiotu Systemy Rozproszone w Katedrze Informatyki Akademii Górniczo - Hutniczej. Świadczy to o jej dużej przydatności w popularyzowaniu koncepcji programowania komponentowego.

Zakładając dalszy rozwój edytorów takich jak *CGE,* rodzi się nadzieja, że w przyszłości model *CCM* będzie się dobrze rozwijał, a jego szerokie wykorzystanie przez środowiska naukowe jak i w zastosowaniach komercyjnych stanie się możliwe.

Dodatek A - opis schematu deskryptora CAD

Poniżej przedstawiamy opis znaczników deskryptora *CAD* i ich właściwości, które są obowiązkowe. Korzeniem deskryptora CAD jest znacznik *componentassembly* zdefiniowany następująco:

```
<!ELEMENT componentassembly
(description?,
componentfiles,
partitioning,
connections?,
extension*
)>
<!ATTLIST componentassembly
  id ID #REQUIRED
  derivedfrom CDATA #IMPLIED>
```

Potrzebne właściwości:
id - identyfikator deskryptora CAD -COMPONENTASSEMBLY_ID

1. Sekcja <componentfiles>

```
<!ELEMENT componentfile
  (
     fileinarchive |
     codebase |
      link
   ) >
<!ATTLIST componentfile id ID #REQUIRED type CDATA #IMPLIED >
```

   Potrzebne właściwości:
   - Atrybut ID w znaczniku componentfile - COMPONENTFILE_ID
   - Atrybut NAME w znaczniku fileinarchive - COMPONENTFILE_FILE
   - Atrybut HREF w znaczniku LINK - COMPONENTFILE_LINK
2. Sekcja <partitioning>

```
<!ELEMENT partitioning
  (
```

```
 homeplacement |
 executableplacement |
 processcollocation |
 hostcollocation |
 extension
)* >
```

a) znacznik <hostcollocation>

```
<!ELEMENT hostcollocation
(
    usagename?,
    impltype?,
    (
       homeplacement |
       executableplacement |
       processcollocation |
       extension
     )+,
    destination? )
>
<!ATTLIST hostcollocation id ID #IMPLIED cardinality CDATA "1" >
```

b) znacznik <homeplacement>

Opisuje sposób alokacji (*deployment*) obiektu *home* komponentu, może występować bezpośrednio w *partitioning* (nie ma żadnych powiązań z alokacją innych obiektów *home*) albo zagnieżdzony w hostcollocation (processcollocation)

```
<!ELEMENT homeplacement
(
    usagename?,
    componentfileref,
    componentimplref?,
    homeproperties?,
    componentproperties?,
    registerwithhomefinder*,
    registerwithnaming*,
    registerwithtrader*,
    componentinstantiation*,
    destination?,
     extension* ) >
<!ATTLIST homeplacement id ID #REQUIRED cardinality CDATA "1" >
```

i. znacznik <componentfileref >
   odwołanie car-a po id zdefiniowanym w sekcji <componentfiles>
ii. znacznik <componentimplref>
   odwołanie do implementacji componentu
iii. znacznik <componentproperties>
   properties defaultowe służy do ustawienia właściwości komponentów tworzonych przez opisywanego obiektu *home*

```
<!ELEMENT componentproperties ( fileinarchive ) >
```

iv. znacznik <registerwithhomefinder>
specyfikuje nazwy pod jakimi home się rejestruje w home finder

```
<!ELEMENT registerwithhomefinder EMPTY >
<!ATTLIST registerwithhomefinder name CDATA #REQUIRED
```

v. znacznik <registerwithnaming>
specyfikuje nazwy pod jakimi home się rejestruje w name service

```
<!ELEMENT registerwithnaming EMPTY >
<!ATTLIST registerwithnaming name CDATA #IMPLIED >
```

vi. znacznik <componentinstantiation>
opis inicjalizacji komponentu przez home'a

```
<!ELEMENT componentinstantiation
(
    usagename?,
    componentproperties?,
     registercomponent*,
     extension*
) >
<!ATTLIST componentinstantiation id ID #REQUIRED >
```

- znacznik <componentproperties>
  odnośnik do opisu własciwości komponentu.

```
<!ELEMENT componentproperties ( fileinarchive |codebase ) >
```

- znacznik <registercomponent>
  Rejestracja instancji komponentu

```
<!ELEMENT registercomponent (
      (
          emitsidentifier |
          providesidentifier |
          publishesidentifier
       )?,
          (
             registerwithnaming |
             registerwithtrader )?
      >
```

— znacznik <registerwithnaming>
Rejestracja w naming service elementu

```
<!ELEMENT registerwithnaming EMPTY >
<!ATTLIST registerwithnaming name CDATA #IMPLIED >
```

Potrzebne właściwości:

- Atrybut ID w znaczniku componentinstantiation - COMPONENTINSTANTIATION_ID
- Nazwa pliku w podznaczniku componentproperties - COMPONENTPROPERTIES_FILE
- LINK w podznaczniku componentproperties - COMPONENTPROPERTIES_LINK

- Name w podznaczniku registercomponent - REGISTERCOMPONENT_NAME

c) znacznik <destination>
określa fizyczne miejsce osadzenia

```
<!ELEMENT destination ( node | ( installation, activation ) ) >
<!ELEMENT node EMPTY>
 <!ATTLIST node name CDATA #REQUIRED>
<!ELEMENT installation ( findby ) >
<!ATTLIST installation type CDATA #IMPLIED>
<!ELEMENT activation ( findby ) >
<!ATTLIST activation type CDATA #IMPLIED>
<!ELEMENT findby (
   namingservice |
   stringifiedobjectref |
   traderquery |
   homefinder |
    extension ) >
```

Potrzebne właściwości:

- Atrybut TYPE w znaczniku installation - INSTALLATION_TYPE,
- Atrybut NAME w znaczniku installation - INSTALLATION_NAME,
- Atrybut TYPE w znaczniku activation - ACTIVATION_TYPE,
- Atrybut NAME w znaczniku activation - ACTIVATION_NAME,
- Atrybut NAME w znaczniku DESTINATION - DESTINATION_NODE.

Potrzebne właściwości znacznika <homeplacement>

- Atrybut IDREF w znaczniku componentimplref - COMPONENTIMPLREF_IDREF,
- Atrybut NAME w znaczniku registerwithhomefinder - REGISTERWITHHOMEFINDER_NAME,
- Atrybut NAME w znaczniku registerwithnaming - REGISTERWITHNAMING_NAME,
- Atrybut ID w znaczniku HOMEPLACEMENT - HOMEPLACEMENT_ID.

Dodatek B - kod źródłowy aplikacji Shape Editor

Autorem zamieszczonego kodu jest twórca edytora *Shapes Editor* - **Elias Volanakis**. Zarówno edytor jak i jego kod są dostępne na prawach *Eclipse Public License v1.0* (http://www.eclipse.org/legal/epl-v10.html). Autorzy pracy dokonali zmian edytorskich (usunięcie komentarzy oraz sekcji import) w celu nadania bardziej zwartej formy prezentowanym listingom kodu.Pełny kod źródłowy zamieszczamy na płycie dołączonej do pracy.

Klasa główna edytora

```
/*******************************************************************************
 * Copyright (c) 2004, 2005 Elias Volanakis and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 * Contributors:
 *    Elias Volanakis - initial API and implementation
 *******************************************************************************/
package org.eclipse.gef.examples.shapes;

/* sekcje "import" potminieto */
public class ShapesEditor extends GraphicalEditorWithFlyoutPalette {

  /** referencja do korzenia modelu, diagramu (grafu) kształtów */
  private ShapesDiagram diagram;

  /** paleta */
  private static PaletteRoot PALETTE_MODEL;

  public ShapesEditor() {
    setEditDomain(new DefaultEditDomain(this));
  }

  protected void configureGraphicalViewer() {
    super.configureGraphicalViewer();

    GraphicalViewer viewer = getGraphicalViewer();
    viewer.setEditPartFactory(new ShapesEditPartFactory());
    viewer.setRootEditPart(new ScalableFreeformRootEditPart());
    viewer.setKeyHandler(new GraphicalViewerKeyHandler(viewer));
    ContextMenuProvider cmProvider = new ShapesEditorContextMenuProvider(
        viewer, getActionRegistry());
    viewer.setContextMenu(cmProvider);
    getSite().registerContextMenu(cmProvider, viewer);
  }

  public void commandStackChanged(EventObject event) {
    firePropertyChange(IEditorPart.PROP_DIRTY);
    super.commandStackChanged(event);
  }

  private void createOutputStream(OutputStream os) throws IOException {
```

```
    ObjectOutputStream oos = new ObjectOutputStream(os);
    oos.writeObject(getModel());
    oos.close();
  }

  protected PaletteViewerProvider createPaletteViewerProvider() {
    return new PaletteViewerProvider(getEditDomain()) {
      protected void configurePaletteViewer(PaletteViewer viewer) {
        super.configurePaletteViewer(viewer);
        viewer
            .addDragSourceListener(new TemplateTransferDragSourceListener(
                viewer));
      }
    };
  }

  private TransferDropTargetListener createTransferDropTargetListener() {
    return new TemplateTransferDropTargetListener(getGraphicalViewer()) {
      protected CreationFactory getFactory(Object template) {
        return new SimpleFactory((Class) template);
      }
    };
  }

public void doSave(IProgressMonitor monitor) {
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  try {
    createOutputStream(out);
    IFile file = ((IFileEditorInput) getEditorInput()).getFile();
    file.setContents(
      new ByteArrayInputStream(out.toByteArray()),
      true,
      false,
      monitor);
    getCommandStack().markSaveLocation();
  } catch (CoreException ce) {
    ce.printStackTrace();
  } catch (IOException ioe) {
    ioe.printStackTrace();
  }
}  public void doSaveAs() {

    Shell shell = getSite().getWorkbenchWindow().getShell();
    SaveAsDialog dialog = new SaveAsDialog(shell);
    dialog.setOriginalFile(((IFileEditorInput) getEditorInput()).getFile());
    dialog.open();
```

```
  IPath path = dialog.getResult();
  if (path != null) {
    final IFile file = ResourcesPlugin.getWorkspace().getRoot()
        .getFile(path);
    try {
      new ProgressMonitorDialog(shell).run(false, false,
          new WorkspaceModifyOperation() {
            public void execute(final IProgressMonitor monitor) {
              try {
                ByteArrayOutputStream out = new ByteArrayOutputStream();
                createOutputStream(out);
                file.create(new ByteArrayInputStream(out
                    .toByteArray()), true, monitor);
              } catch (CoreException ce) {
                ce.printStackTrace();
              } catch (IOException ioe) {
                ioe.printStackTrace();
              }
            }
          });
      setInput(new FileEditorInput(file));
      getCommandStack().markSaveLocation();
    } catch (InterruptedException ie) {
      ie.printStackTrace();
    } catch (InvocationTargetException ite) {
      ite.printStackTrace();
    }
  }
}

public Object getAdapter(Class type) {
  if (type == IContentOutlinePage.class)
    return new ShapesOutlinePage(new TreeViewer());
  return super.getAdapter(type);
}

ShapesDiagram getModel() {
  return diagram;
}

protected FlyoutPreferences getPalettePreferences() {
  return ShapesEditorPaletteFactory.createPalettePreferences();
}

protected PaletteRoot getPaletteRoot() {
```

```
    if (PALETTE_MODEL == null)
      PALETTE_MODEL = ShapesEditorPaletteFactory.createPalette();
    return PALETTE_MODEL;
  }

  private void handleLoadException(Exception e) {
    System.err.println("** Load failed. Using default model. **");
    e.printStackTrace();
    diagram = new ShapesDiagram();
  }

  protected void initializeGraphicalViewer() {
    super.initializeGraphicalViewer();
    GraphicalViewer viewer = getGraphicalViewer();
    viewer.setContents(getModel());
    viewer.addDropTargetListener(createTransferDropTargetListener());
  }


  public boolean isSaveAsAllowed() {
    return true;
  }

  protected void setInput(IEditorInput input) {
    super.setInput(input);
    try {
      IFile file = ((IFileEditorInput) input).getFile();
      ObjectInputStream in = new ObjectInputStream(file.getContents());
      diagram = (ShapesDiagram) in.readObject();
      in.close();
      setPartName(file.getName());
    } catch (IOException e) {
      handleLoadException(e);
    } catch (CoreException e) {
      handleLoadException(e);
    } catch (ClassNotFoundException e) {
      handleLoadException(e);
    }
  }


  public class ShapesOutlinePage extends ContentOutlinePage {

    public ShapesOutlinePage(EditPartViewer viewer) {
      super(viewer);
    }
```

```
    public void createControl(Composite parent) {
      getViewer().createControl(parent);
      getViewer().setEditDomain(getEditDomain());
      getViewer().setEditPartFactory(new ShapesTreeEditPartFactory());
      ContextMenuProvider cmProvider = new ShapesEditorContextMenuProvider(
          getViewer(), getActionRegistry());
      getViewer().setContextMenu(cmProvider);
      getSite().registerContextMenu(
          "org.eclipse.gef.examples.shapes.outline.contextmenu",
          cmProvider, getSite().getSelectionProvider());
      getSelectionSynchronizer().addViewer(getViewer());

      getViewer().setContents(getModel());

    }

    public void dispose() {

      getSelectionSynchronizer().removeViewer(getViewer());
      super.dispose();
    }

    public Control getControl() {
      return getViewer().getControl();
    }


    public void init(IPageSite pageSite) {
      super.init(pageSite);
      ActionRegistry registry = getActionRegistry();
      IActionBars bars = pageSite.getActionBars();
      String id = ActionFactory.UNDO.getId();
      bars.setGlobalActionHandler(id, registry.getAction(id));
      id = ActionFactory.REDO.getId();
      bars.setGlobalActionHandler(id, registry.getAction(id));
      id = ActionFactory.DELETE.getId();
      bars.setGlobalActionHandler(id, registry.getAction(id));
    }
  }

}
```

Klasy modelu

Klasa Connection

```
package org.eclipse.gef.examples.shapes.model;

import org.eclipse.ui.views.properties.ComboBoxPropertyDescriptor;
import org.eclipse.ui.views.properties.IPropertyDescriptor;

import org.eclipse.draw2d.Graphics;

public class Connection extends ModelElement {

public static final Integer SOLID_CONNECTION = new Integer(Graphics.LINE_SOLID);

public static final Integer DASHED_CONNECTION = new Integer(Graphics.LINE_DASH);

public static final String LINESTYLE_PROP = "LineStyle";
private static final IPropertyDescriptor[] descriptors =
                                          new IPropertyDescriptor[1];
private static final String SOLID_STR = "Solid";
private static final String DASHED_STR = "Dashed";
private static final long serialVersionUID = 1;

private boolean isConnected;
private int lineStyle = Graphics.LINE_SOLID;
private Shape source;
private Shape target;

static {
  descriptors[0] = new ComboBoxPropertyDescriptor(LINESTYLE_PROP, LINESTYLE_PROP,
      new String[] {SOLID_STR, DASHED_STR});
}

public Connection(Shape source, Shape target) {
  reconnect(source, target);
}

public void disconnect() {
  if (isConnected) {
    source.removeConnection(this);
    target.removeConnection(this);
    isConnected = false;
  }
}
```

```
public int getLineStyle() {
  return lineStyle;
}

public IPropertyDescriptor[] getPropertyDescriptors() {
  return descriptors;
}

public Object getPropertyValue(Object id) {
  if (id.equals(LINESTYLE_PROP)) {
    if (getLineStyle() == Graphics.LINE_DASH)
      return new Integer(1);
    return new Integer(0);
  }
  return super.getPropertyValue(id);
}

public Shape getSource() {
  return source;
}

public Shape getTarget() {
  return target;
}

public void reconnect() {
  if (!isConnected) {
    source.addConnection(this);
    target.addConnection(this);
    isConnected = true;
  }
}

public void reconnect(Shape newSource, Shape newTarget) {
  if (newSource == null || newTarget == null || newSource == newTarget) {
    throw new IllegalArgumentException();
  }
  disconnect();
  this.source = newSource;
  this.target = newTarget;
  reconnect();
}

public void setLineStyle(int lineStyle) {
  if (lineStyle != Graphics.LINE_DASH && lineStyle != Graphics.LINE_SOLID) {
    throw new IllegalArgumentException();
```

```
  }
  this.lineStyle = lineStyle;
  firePropertyChange(LINESTYLE_PROP, null, new Integer(this.lineStyle));
}

public void setPropertyValue(Object id, Object value) {
  if (id.equals(LINESTYLE_PROP))
    setLineStyle(new Integer(1).equals(value)
        ? Graphics.LINE_DASH : Graphics.LINE_SOLID);
  else
    super.setPropertyValue(id, value);
}

}
```

Klasa ShapesDiagram

```
package org.eclipse.gef.examples.shapes.model;
/** pominiento sekcje import*/
public class ShapesDiagram extends ModelElement {

public static final String CHILD_ADDED_PROP = "ShapesDiagram.ChildAdded";
public static final String CHILD_REMOVED_PROP = "ShapesDiagram.ChildRemoved";
private List shapes = new ArrayList();

public boolean addChild(Shape s) {
  if (s != null && shapes.add(s)) {
    firePropertyChange(CHILD_ADDED_PROP, null, s);
    return true;
  }
  return false;
}

public List getChildren() {
  return shapes;
}

public boolean removeChild(Shape s) {
  if (s != null && shapes.remove(s)) {
    firePropertyChange(CHILD_REMOVED_PROP, null, s);
    return true;
  }
  return false;
}
```

```
}
```

Klasa ModelElement

```
package org.eclipse.gef.examples.shapes.model;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.Serializable;

import org.eclipse.ui.views.properties.IPropertyDescriptor;
import org.eclipse.ui.views.properties.IPropertySource;

public abstract class ModelElement implements IPropertySource, Serializable {

private static final IPropertyDescriptor[] EMPTY_ARRAY =
                              new IPropertyDescriptor[0];

private static final long serialVersionUID = 1;

private transient PropertyChangeSupport pcsDelegate =
                               new PropertyChangeSupport(this);

public synchronized void addPropertyChangeListener(PropertyChangeListener l) {
  if (l == null) {
    throw new IllegalArgumentException();
  }
  pcsDelegate.addPropertyChangeListener(l);
}

protected void firePropertyChange(String property, Object oldValue,
                                  Object newValue) {
  if (pcsDelegate.hasListeners(property)) {
    pcsDelegate.firePropertyChange(property, oldValue, newValue);
  }
}

public Object getEditableValue() {
  return this;
}

public IPropertyDescriptor[] getPropertyDescriptors() {
```

```
  return EMPTY_ARRAY;
}

public Object getPropertyValue(Object id) {
  return null;
}

public boolean isPropertySet(Object id) {
  return false;
}

private void readObject(ObjectInputStream in)
throws IOException, ClassNotFoundException {
  in.defaultReadObject();
  pcsDelegate = new PropertyChangeSupport(this);
}

public synchronized void removePropertyChangeListener(PropertyChangeListener l) {
  if (l != null) {
    pcsDelegate.removePropertyChangeListener(l);
  }
}

public void resetPropertyValue(Object id) {}

public void setPropertyValue(Object id, Object value) {}
}
```

Klasa Shape

```
package org.eclipse.gef.examples.shapes.model;
/** pominiento sekcje import*/
public abstract class Shape extends ModelElement {

private static IPropertyDescriptor[] descriptors;
private static final String HEIGHT_PROP = "Shape.Height";
public static final String LOCATION_PROP = "Shape.Location";
public static final String SIZE_PROP = "Shape.Size";
public static final String SOURCE_CONNECTIONS_PROP = "Shape.SourceConn";
public static final String TARGET_CONNECTIONS_PROP = "Shape.TargetConn";
private static final String WIDTH_PROP = "Shape.Width";
private static final String XPOS_PROP = "Shape.xPos";
private static final String YPOS_PROP = "Shape.yPos";
```

```
static {
  descriptors = new IPropertyDescriptor[] {
      new TextPropertyDescriptor(XPOS_PROP, "X"),
      new TextPropertyDescriptor(YPOS_PROP, "Y"),
      new TextPropertyDescriptor(WIDTH_PROP, "Width"),
      new TextPropertyDescriptor(HEIGHT_PROP, "Height"),
  };
  for (int i = 0; i < descriptors.length; i++) {
    ((PropertyDescriptor) descriptors[i]).setValidator(new ICellEditorValidator() {
      public String isValid(Object value) {
        int intValue = -1;
        try {
          intValue = Integer.parseInt((String) value);
        } catch (NumberFormatException exc) {
          return "Not a number";
        }
        return (intValue >= 0) ? null : "Value must be >=  0";
      }
    });
  }
}
private Point location = new Point(0, 0);
private Dimension size = new Dimension(50, 50);

private List sourceConnections = new ArrayList();
private List targetConnections = new ArrayList();


void addConnection(Connection conn) {
  if (conn == null || conn.getSource() == conn.getTarget()) {
    throw new IllegalArgumentException();
  }
  if (conn.getSource() == this) {
    sourceConnections.add(conn);
    firePropertyChange(SOURCE_CONNECTIONS_PROP, null, conn);
  } else if (conn.getTarget() == this) {
    targetConnections.add(conn);
    firePropertyChange(TARGET_CONNECTIONS_PROP, null, conn);
  }
}


public Point getLocation() {
  return location.getCopy();
}
```

```
public IPropertyDescriptor[] getPropertyDescriptors() {
  return descriptors;
}

public Object getPropertyValue(Object propertyId) {
  if (XPOS_PROP.equals(propertyId)) {
    return Integer.toString(location.x);
  }
  if (YPOS_PROP.equals(propertyId)) {
    return Integer.toString(location.y);
  }
  if (HEIGHT_PROP.equals(propertyId)) {
    return Integer.toString(size.height);
  }
  if (WIDTH_PROP.equals(propertyId)) {
    return Integer.toString(size.width);
  }
  return super.getPropertyValue(propertyId);
}

public Dimension getSize() {
  return size.getCopy();
}

public List getSourceConnections() {
  return new ArrayList(sourceConnections);
}

public List getTargetConnections() {
  return new ArrayList(targetConnections);
}

void removeConnection(Connection conn) {
  if (conn == null) {
    throw new IllegalArgumentException();
  }
  if (conn.getSource() == this) {
    sourceConnections.remove(conn);
    firePropertyChange(SOURCE_CONNECTIONS_PROP, null, conn);
  } else if (conn.getTarget() == this) {
    targetConnections.remove(conn);
    firePropertyChange(TARGET_CONNECTIONS_PROP, null, conn);
  }
}

public void setLocation(Point newLocation) {
```

```
  if (newLocation == null) {
    throw new IllegalArgumentException();
  }
  location.setLocation(newLocation);
  firePropertyChange(LOCATION_PROP, null, location);
}

public void setPropertyValue(Object propertyId, Object value) {
  if (XPOS_PROP.equals(propertyId)) {
    int x = Integer.parseInt((String) value);
    setLocation(new Point(x, location.y));
  } else if (YPOS_PROP.equals(propertyId)) {
    int y = Integer.parseInt((String) value);
    setLocation(new Point(location.x, y));
  } else if (HEIGHT_PROP.equals(propertyId)) {
    int height = Integer.parseInt((String) value);
    setSize(new Dimension(size.width, height));
  } else if (WIDTH_PROP.equals(propertyId)) {
    int width = Integer.parseInt((String) value);
    setSize(new Dimension(width, size.height));
  } else {
    super.setPropertyValue(propertyId, value);
  }
}

public void setSize(Dimension newSize) {
  if (newSize != null) {
    size.setSize(newSize);
    firePropertyChange(SIZE_PROP, null, size);
  }
}
}
```

Klasa EllipticalShape

```
package org.eclipse.gef.examples.shapes.model;

import org.eclipse.swt.graphics.Image;

public class EllipticalShape extends Shape {

/** A 16x16 pictogram of an elliptical shape. */
private static final Image ELLIPSE_ICON = createImage("icons/ellipse16.gif");
```

```
private static final long serialVersionUID = 1;

public Image getIcon() {
  return ELLIPSE_ICON;
}

public String toString() {
  return "Ellipse " + hashCode();
}
}
```

Klasa RectangularShape

```
package org.eclipse.gef.examples.shapes.model;

import org.eclipse.swt.graphics.Image;

public class RectangularShape extends Shape {
private static final Image RECTANGLE_ICON = createImage("icons/rectangle16.gif");

private static final long serialVersionUID = 1;

public Image getIcon() {
  return RECTANGLE_ICON;
}

public String toString() {
  return "Rectangle " + hashCode();
}
}
```

Klasy kontrolera

Klasa ConnectionEditPart

```
package org.eclipse.gef.examples.shapes.parts;

/** sekcje import pominieto*/
class ConnectionEditPart extends AbstractConnectionEditPart
  implements PropertyChangeListener {

public void activate() {
  if (!isActive()) {
    super.activate();
    ((ModelElement) getModel()).addPropertyChangeListener(this);
  }
}

protected void createEditPolicies() {
  installEditPolicy(EditPolicy.CONNECTION_ENDPOINTS_ROLE,
      new ConnectionEndpointEditPolicy());
    installEditPolicy(EditPolicy.CONNECTION_ROLE, new ConnectionEditPolicy() {
    protected Command getDeleteCommand(GroupRequest request) {
      return new ConnectionDeleteCommand(getCastedModel());
    }
  });
}

protected IFigure createFigure() {
  PolylineConnection connection = (PolylineConnection) super.createFigure();
  connection.setTargetDecoration(new PolygonDecoration());
  connection.setLineStyle(getCastedModel() .getLineStyle());
  return connection;
}

public void deactivate() {
  if (isActive()) {
    super.deactivate();
    ((ModelElement) getModel()).removePropertyChangeListener(this);
  }
}

private Connection getCastedModel() {
  return (Connection) getModel();
}
```

```
public void propertyChange(PropertyChangeEvent event) {
  String property = event.getPropertyName();
  if (Connection.LINESTYLE_PROP.equals(property)) {
    ((PolylineConnection) getFigure()).setLineStyle(getCastedModel()
    .getLineStyle());
  }
}

}
```

Klasa DiagramEditPart

```
package org.eclipse.gef.examples.shapes.parts;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.util.List;

/** sekcje import pominieto*/
class DiagramEditPart extends AbstractGraphicalEditPart
  implements PropertyChangeListener  {

public void activate() {
  if (!isActive()) {
    super.activate();
    ((ModelElement) getModel()).addPropertyChangeListener(this);
  }
}


protected void createEditPolicies() {
  installEditPolicy(EditPolicy.COMPONENT_ROLE, new RootComponentEditPolicy());
  installEditPolicy(EditPolicy.LAYOUT_ROLE,  new ShapesXYLayoutEditPolicy());
}

protected IFigure createFigure() {
  Figure f = new FreeformLayer();
  f.setBorder(new MarginBorder(3));
  f.setLayoutManager(new FreeformLayout());
  ConnectionLayer connLayer =
     (ConnectionLayer)getLayer(LayerConstants.CONNECTION_LAYER);
  connLayer.setConnectionRouter(new ShortestPathConnectionRouter(f));
  return f;
}
```

```
public void deactivate() {
  if (isActive()) {
    super.deactivate();
    ((ModelElement) getModel()).removePropertyChangeListener(this);
  }
}

private ShapesDiagram getCastedModel() {
  return (ShapesDiagram) getModel();
}

protected List getModelChildren() {
  return getCastedModel().getChildren();
}

public void propertyChange(PropertyChangeEvent evt) {
  String prop = evt.getPropertyName();
  if (ShapesDiagram.CHILD_ADDED_PROP.equals(prop)
      || ShapesDiagram.CHILD_REMOVED_PROP.equals(prop)) {
    refreshChildren();
  }
}
}
```

Klasa DiagramTreeEditPart

```
package org.eclipse.gef.examples.shapes.parts;
/** pominieto sekcje import*/
class DiagramTreeEditPart extends AbstractTreeEditPart
  implements PropertyChangeListener {

DiagramTreeEditPart(ShapesDiagram model) {
  super(model);
}

public void activate() {
  if (!isActive()) {
    super.activate();
    ((ModelElement) getModel()).addPropertyChangeListener(this);
  }
}

protected void createEditPolicies() {
  if (getParent() instanceof RootEditPart) {
```

```
    installEditPolicy(EditPolicy.COMPONENT_ROLE, new RootComponentEditPolicy());
  }
}

public void deactivate() {
  if (isActive()) {
    super.deactivate();
    ((ModelElement) getModel()).removePropertyChangeListener(this);
  }
}

private ShapesDiagram getCastedModel() {
  return (ShapesDiagram) getModel();
}

private EditPart getEditPartForChild(Object child) {
  return (EditPart) getViewer().getEditPartRegistry().get(child);
}

protected List getModelChildren() {
  return getCastedModel().getChildren();
}

public void propertyChange(PropertyChangeEvent evt) {
  String prop = evt.getPropertyName();
  if (ShapesDiagram.CHILD_ADDED_PROP.equals(prop)) {
    addChild(createChild(evt.getNewValue()), -1);
  } else if (ShapesDiagram.CHILD_REMOVED_PROP.equals(prop)) {
      removeChild(getEditPartForChild(evt.getNewValue()));
  } else {
    refreshVisuals();
  }
}
}
```

Klasa ShapeEditPart

```
package org.eclipse.gef.examples.shapes.parts;
/** pominieto sekcje import*/
class ShapeEditPart extends AbstractGraphicalEditPart
  implements PropertyChangeListener, NodeEditPart {

private ConnectionAnchor anchor;

public void activate() {
```

```
  if (!isActive()) {
    super.activate();
    ((ModelElement) getModel()).addPropertyChangeListener(this);
  }
}


protected void createEditPolicies() {
  installEditPolicy(EditPolicy.COMPONENT_ROLE, new ShapeComponentEditPolicy());
  /** implementacja polityki GraphicalNodeEditPolicy*/
  installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE, new GraphicalNodeEditPolicy() {

    protected Command getConnectionCompleteCommand
         (CreateConnectionRequest request) {
      ConnectionCreateCommand cmd
        = (ConnectionCreateCommand) request.getStartCommand();
      cmd.setTarget((Shape) getHost().getModel());
      return cmd;
    }

    protected Command getConnectionCreateCommand(CreateConnectionRequest request) {
      Shape source = (Shape) getHost().getModel();
      int style = ((Integer) request.getNewObjectType()).intValue();
      ConnectionCreateCommand cmd = new ConnectionCreateCommand(source, style);
      request.setStartCommand(cmd);
      return cmd;
    }
    /
    protected Command getReconnectSourceCommand(ReconnectRequest request) {
      Connection conn = (Connection) request.getConnectionEditPart().getModel();
      Shape newSource = (Shape) getHost().getModel();
      ConnectionReconnectCommand cmd = new ConnectionReconnectCommand(conn);
      cmd.setNewSource(newSource);
      return cmd;
    }

    protected Command getReconnectTargetCommand(ReconnectRequest request) {
      Connection conn = (Connection) request.getConnectionEditPart().getModel();
      Shape newTarget = (Shape) getHost().getModel();
      ConnectionReconnectCommand cmd = new ConnectionReconnectCommand(conn);
      cmd.setNewTarget(newTarget);
      return cmd;
    }
  });
}
```

```
protected IFigure createFigure() {
  IFigure f = createFigureForModel();
  f.setOpaque(true);
  f.setBackgroundColor(ColorConstants.green);
  return f;
}


private IFigure createFigureForModel() {
  if (getModel() instanceof EllipticalShape) {
    return new Ellipse();
  } else if (getModel() instanceof RectangularShape) {
    return new RectangleFigure();
  } else {
    throw new IllegalArgumentException();
  }
}

public void deactivate() {
  if (isActive()) {
    super.deactivate();
    ((ModelElement) getModel()).removePropertyChangeListener(this);
  }
}

private Shape getCastedModel() {
  return (Shape) getModel();
}

protected ConnectionAnchor getConnectionAnchor() {
  if (anchor == null) {
    if (getModel() instanceof EllipticalShape)
      anchor = new EllipseAnchor(getFigure());
    else if (getModel() instanceof RectangularShape)
      anchor = new ChopboxAnchor(getFigure());
    else
      throw new IllegalArgumentException("unexpected model");
  }
  return anchor;
}

protected List getModelSourceConnections() {
  return getCastedModel().getSourceConnections();
}

protected List getModelTargetConnections() {
```

```
  return getCastedModel().getTargetConnections();
}

public ConnectionAnchor getSourceConnectionAnchor(ConnectionEditPart connection) {
  return getConnectionAnchor();
}

public ConnectionAnchor getSourceConnectionAnchor(Request request) {
  return getConnectionAnchor();
}

public ConnectionAnchor getTargetConnectionAnchor(ConnectionEditPart connection) {
  return getConnectionAnchor();
}

public ConnectionAnchor getTargetConnectionAnchor(Request request) {
  return getConnectionAnchor();
}

public void propertyChange(PropertyChangeEvent evt) {
  String prop = evt.getPropertyName();
  if (Shape.SIZE_PROP.equals(prop) || Shape.LOCATION_PROP.equals(prop)) {
    refreshVisuals();
  } else if (Shape.SOURCE_CONNECTIONS_PROP.equals(prop)) {
    refreshSourceConnections();
  } else if (Shape.TARGET_CONNECTIONS_PROP.equals(prop)) {
    refreshTargetConnections();
  }
}

protected void refreshVisuals() {
  Rectangle bounds = new Rectangle(getCastedModel().getLocation(),
      getCastedModel().getSize());
  ((GraphicalEditPart) getParent()).setLayoutConstraint(this, getFigure(),
                    bounds);
}
}
```

Klasa ShapeTreeEditPart

```
package org.eclipse.gef.examples.shapes.parts;
/** pominieto sekcje import*/
class ShapeTreeEditPart extends AbstractTreeEditPart implements
    PropertyChangeListener {
```

```
ShapeTreeEditPart(Shape model) {
  super(model);
}

public void activate() {
  if (!isActive()) {
    super.activate();
    ((ModelElement) getModel()).addPropertyChangeListener(this);
  }
}

protected void createEditPolicies() {
  installEditPolicy(EditPolicy.COMPONENT_ROLE, new ShapeComponentEditPolicy());
}

public void deactivate() {
  if (isActive()) {
    super.deactivate();
    ((ModelElement) getModel()).removePropertyChangeListener(this);
  }
}

private Shape getCastedModel() {
  return (Shape) getModel();
}

protected Image getImage() {
  return getCastedModel().getIcon();
}

protected String getText() {
  return getCastedModel().toString();
}

public void propertyChange(PropertyChangeEvent evt) {
  refreshVisuals(); //
}
}
```

Fabryki elementów kontrolera

Klasa ShapesEditPartFactory

```
package org.eclipse.gef.examples.shapes.parts;
/** pominieto sekcje import*/
public class ShapesEditPartFactory implements EditPartFactory {

public EditPart createEditPart(EditPart context, Object modelElement) {
  EditPart part = getPartForElement(modelElement);
  part.setModel(modelElement);
  return part;
}

private EditPart getPartForElement(Object modelElement) {
  if (modelElement instanceof ShapesDiagram)
    return new DiagramEditPart();
  if (modelElement instanceof Shape)
    return new ShapeEditPart();
  if (modelElement instanceof Connection)
    return new ConnectionEditPart();
  throw new RuntimeException(
      "Can't create part for model element: "
      + ((modelElement != null) ?
          modelElement.getClass().getName() : "null"));
}
}
```

Klasa ShapesTreeEditPartFactory

```
package org.eclipse.gef.examples.shapes.parts;
/** pominieto sekcje import*/
public class ShapesTreeEditPartFactory implements EditPartFactory {

public EditPart createEditPart(EditPart context, Object model) {
  if (model instanceof Shape) {
    return new ShapeTreeEditPart((Shape) model);
  }
  if (model instanceof ShapesDiagram) {
    return new DiagramTreeEditPart((ShapesDiagram) model);
  }
  return null;

}
```

Klasy polityk

Klasa ShapeComponentEditPolicy

```
package org.eclipse.gef.examples.shapes.parts;
/** pominieto sekcje import*/
class ShapeComponentEditPolicy extends ComponentEditPolicy {
protected Command createDeleteCommand(GroupRequest deleteRequest) {
  Object parent = getHost().getParent().getModel();
  Object child = getHost().getModel();
  if (parent instanceof ShapesDiagram && child instanceof Shape) {
    return new ShapeDeleteCommand((ShapesDiagram) parent, (Shape) child);
  }
  return super.createDeleteCommand(deleteRequest);
} }
```

Klasa ShapesXYLayoutEditPolicy

```
/** implementacja polityki XYLayoutEditPolicy */
private static class ShapesXYLayoutEditPolicy extends XYLayoutEditPolicy {

  protected Command createAddCommand(EditPart child, Object constraint) {
    return null;   }

  protected Command createChangeConstraintCommand(ChangeBoundsRequest request,
      EditPart child, Object constraint) {
    if (child instanceof ShapeEditPart && constraint instanceof Rectangle) {
      return new ShapeSetConstraintCommand(
          (Shape) child.getModel(), request, (Rectangle) constraint);
    }
    return super.createChangeConstraintCommand(request, child, constraint);
  }

  protected Command createChangeConstraintCommand(EditPart child,
      Object constraint) {     return null;   }

  protected Command getCreateCommand(CreateRequest request) {
    Object childClass = request.getNewObjectType();
    if (childClass == EllipticalShape.class ||
        childClass == RectangularShape.class) {
      return new ShapeCreateCommand((Shape)request.getNewObject(),
          (ShapesDiagram)getHost().getModel(),
          (Rectangle)getConstraintFor(request));
    }
    return null;
  }
```

```
  protected Command getDeleteDependantCommand(Request request) {
   return null;   }
}
```

Klasy komend

Klasa ConnectionCreateCommand

```
package org.eclipse.gef.examples.shapes.model.commands;
/** pominieto sekcje import*/
public class ConnectionCreateCommand extends Command {

private Connection connection;
private final int lineStyle;
private final Shape source;
private Shape target;

public ConnectionCreateCommand(Shape source, int lineStyle) {
  if (source == null) {
    throw new IllegalArgumentException();
  }
  setLabel("connection creation");
  this.source = source;
  this.lineStyle = lineStyle;
}

public boolean canExecute() {
  if (source.equals(target)) {
    return false;
  }
  for (Iterator iter = source.getSourceConnections().iterator(); iter.hasNext();) {
    Connection conn = (Connection) iter.next();
    if (conn.getTarget().equals(target)) {
      return false;
    }
  }
  return true;
}

public void execute() {
  connection = new Connection(source, target);
  connection.setLineStyle(lineStyle);
}

public void redo() {
  connection.reconnect();
}

public void setTarget(Shape target) {
  if (target == null) {
```

```
    throw new IllegalArgumentException();
  }
  this.target = target;
}

public void undo() {
  connection.disconnect();
}
}
```

Klasa ConnectionDeleteCommand

```
package org.eclipse.gef.examples.shapes.model.commands;
/** pominieto sekcje import*/
public class ConnectionDeleteCommand extends Command {
private final Connection connection;

public ConnectionDeleteCommand(Connection conn) {
  if (conn == null) {
    throw new IllegalArgumentException();
  }
  setLabel("connection deletion");
  this.connection = conn;
}

public void execute() {
  connection.disconnect();
}

public void undo() {
  connection.reconnect();
}
}
```

Klasa ConnectionReconnectCommand

```
package org.eclipse.gef.examples.shapes.model.commands;
/** pominieto sekcje import*/
public class ConnectionReconnectCommand extends Command {

private Connection connection;
private Shape newSource;
private Shape newTarget;
```

```
private final Shape oldSource;
private final Shape oldTarget;

public ConnectionReconnectCommand(Connection conn) {
  if (conn == null) {
    throw new IllegalArgumentException();
  }
  this.connection = conn;
  this.oldSource = conn.getSource();
  this.oldTarget = conn.getTarget();
}

public boolean canExecute() {
  if (newSource != null) {
    return checkSourceReconnection();
  } else if (newTarget != null) {
    return checkTargetReconnection();
  }
  return false;
}

private boolean checkSourceReconnection() {
  if (newSource.equals(oldTarget)) {
    return false;
  }
  for (Iterator iter = newSource.getSourceConnections().iterator();
              iter.hasNext();) {
    Connection conn = (Connection) iter.next();
    if (conn.getTarget().equals(oldTarget) &&  !conn.equals(connection)) {
      return false;
    }
  }
  return true;
}

private boolean checkTargetReconnection() {
  if (newTarget.equals(oldSource)) {
    return false;
  }
  for (Iterator iter = newTarget.getTargetConnections().iterator();
              iter.hasNext();) {
    Connection conn = (Connection) iter.next();
    if (conn.getSource().equals(oldSource) && !conn.equals(connection)) {
      return false;
    }
  }
```

```
  return true;
}

public void execute() {
  if (newSource != null) {
    connection.reconnect(newSource, oldTarget);
  } else if (newTarget != null) {
    connection.reconnect(oldSource, newTarget);
  } else {
    throw new IllegalStateException("Should not happen");
  }
}

public void setNewSource(Shape connectionSource) {
  if (connectionSource == null) {
    throw new IllegalArgumentException();
  }
  setLabel("move connection startpoint");
  newSource = connectionSource;
  newTarget = null;
}

public void setNewTarget(Shape connectionTarget) {
  if (connectionTarget == null) {
    throw new IllegalArgumentException();
  }
  setLabel("move connection endpoint");
  newSource = null;
  newTarget = connectionTarget;
}

public void undo() {
  connection.reconnect(oldSource, oldTarget);
}
}
```

Klasa ShapeCreateCommand

```
package org.eclipse.gef.examples.shapes.model.commands;
/** pominieto sekcje import*/cd
public class ShapeCreateCommand
  extends Command
{
private Shape newShape;
private final ShapesDiagram parent;
```

```
private Rectangle bounds;

public ShapeCreateCommand(Shape newShape, ShapesDiagram parent, Rectangle bounds) {
  this.newShape = newShape;
  this.parent = parent;
  this.bounds = bounds;
  setLabel("shape creation");
}

public boolean canExecute() {
  return newShape != null && parent != null && bounds != null;
}

public void execute() {
  newShape.setLocation(bounds.getLocation());
  Dimension size = bounds.getSize();
  if (size.width > 0 && size.height > 0)
    newShape.setSize(size);
  redo();
}

public void redo() {
  parent.addChild(newShape);
}

public void undo() {
  parent.removeChild(newShape);
}
}
```

Klasa ShapeDeleteCommand

```
package org.eclipse.gef.examples.shapes.model.commands;
/** pominieto sekcje import*/
public class ShapeDeleteCommand extends Command {
private final Shape child;
private final ShapesDiagram parent;
private List sourceConnections;
private List targetConnections;
private boolean wasRemoved;

public ShapeDeleteCommand(ShapesDiagram parent, Shape child) {
  if (parent == null || child == null) {
    throw new IllegalArgumentException();
  }
```

```
  setLabel("shape deletion");
  this.parent = parent;
  this.child = child;
}

private void addConnections(List connections) {
  for (Iterator iter = connections.iterator(); iter.hasNext();) {
    Connection conn = (Connection) iter.next();
    conn.reconnect();
  }
}

public boolean canUndo() {
  return wasRemoved;
}

public void execute() {
  sourceConnections = child.getSourceConnections();
  targetConnections = child.getTargetConnections();
  redo();
}

public void redo() {
  wasRemoved = parent.removeChild(child);
  if (wasRemoved) {
    removeConnections(sourceConnections);
    removeConnections(targetConnections);
  }
}

private void removeConnections(List connections) {
  for (Iterator iter = connections.iterator(); iter.hasNext();) {
    Connection conn = (Connection) iter.next();
    conn.disconnect();
  }
}

public void undo() {
  if (parent.addChild(child)) {
    addConnections(sourceConnections);
    addConnections(targetConnections);
  }
}
}
```

Klasa ShapeSetConstraintCommand

```
package org.eclipse.gef.examples.shapes.model.commands;
/** pominieto sekcje import*/
public class ShapeSetConstraintCommand extends Command {
private final Rectangle newBounds;
private Rectangle oldBounds;
private final ChangeBoundsRequest request;
private final Shape shape;

public ShapeSetConstraintCommand(Shape shape, ChangeBoundsRequest req,
    Rectangle newBounds) {
  if (shape == null || req == null || newBounds == null) {
    throw new IllegalArgumentException();
  }
  this.shape = shape;
  this.request = req;
  this.newBounds = newBounds.getCopy();
  setLabel("move / resize");
}

public boolean canExecute() {
  Object type = request.getType();
  return (RequestConstants.REQ_MOVE.equals(type)
      || RequestConstants.REQ_MOVE_CHILDREN.equals(type)
      || RequestConstants.REQ_RESIZE.equals(type)
      || RequestConstants.REQ_RESIZE_CHILDREN.equals(type));
}

public void execute() {
  oldBounds = new Rectangle(shape.getLocation(), shape.getSize());
  redo();
}

public void redo() {
  shape.setSize(newBounds.getSize());
  shape.setLocation(newBounds.getLocation());
}

public void undo() {
  shape.setSize(oldBounds.getSize());
  shape.setLocation(oldBounds.getLocation());
}
}
```

Bibliografia

[1] Philippe Merle, Sylvain Leblanc, Mathieu Vadet, Frank Pilhofer, Tom Ritter, Harald Böhme: *Corba Component Model Tutorial*, OMG Meeting Yokohama, Japonia : 2002.

[2] *CORBA Components*, OMG Document formal/02-06-65, czerwiec 2002

[3] *Naming Service Specification*, OMG Document formal/04-10-03, październik 2004

[4] *The Interface Repository*, OMG Document formal/02-06-46, lipiec 2002

[5] *http://www.eclipse.org/gef/overview.html*, Wprowadzenie do GEF

[6] *http://openccm.objectweb.org/*, strona domowa OpenCCM

[7] S. Bodoff, E. Armstrong, J. Ball, D. B. Carson: *J2EE - Vademecum profesjonalisty*: 2005

[8] *http://www.eclipse.org/emf*, Strona projektu EMF

[9] *http://www.eclipse.org/uml2*, Strona projektu UML

[10] *http://www.omg.org/cgi-bin/doc?formal/05-07-06*, Specyfikacja UML 2.0 dla CCM - formal/05-07-06 (UML Profile for CCM

[11] *http://www.dre.vanderbilt.edu/cosmic/html/overview.shtm*, Informacje na temat CoSMIC

[12] *http://www.ni.com/labview/*, Strona środowiska LabView

[13] *http://www.eclipse.org/articles/Article-Your%20FirstPlug-in/YourFirstPlugin.html* - wprowadzenie do tworzenia wtyczek w środowisku Eclipse

[14] *http://www.webdeveloper.pl/eclipse__otwarte_srodowisko_deweloperskie,323,1,1,pl.html*, Eclipse - otwarte środowisko deweloperskie

[15] Randy Hudson, Patrik Shah: *Tutorial #23: GEF In Depth*, 2005

[16] Koen Aers: *A Gentle Introduction to GEF*, 2005

[17] *http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html* - przewodnik jak stworzyć prosty edytor w środowisku GEF

[18] *http://www.omg.org/cgi-bin/doc?formal/06-04-01* - specyfikacja CCM w wersji 4.0

[19] *http://www-128.ibm.com/developerworks/webservices/library/co-cjct6/* - wprowadzenie do modelu CCM