Czym jest UML

Unified Modeling Language (UML) to graficzny język do obrazowania, specyfikowania, tworzenia i dokumentowania elementów systemów informatycznych. Umożliwia standaryzację sposobu prezentowania systemu z perspektywy procesów biznesowych, funkcji systemowych, klas programistycznych oraz baz danych.

Historia

Języki modelowania obiektowego pojawiły się na przełomie lat siedemdziesiątych i osiemdziesiątych, w związku z powstaniem nowego paradygmatu programowania. Na początku lat 90-tych istniało ponad 50 różnych metod, co prowadziło do problemów ze znalezieniem odpowiedniego języka oraz zrozumieniem projektów opisanych w różny sposób.

Opracowania nowe generacje metod, które miały standaryzować modelowanie obiektowe. Największą popularność zyskały 3 takie metody: Boocha, OOSE Jacobsona (Object-Oriented Software Engineering) oraz OMT Rambaugha (Object Modeling Technique). Każda z metod miała swoje mocne i słabe strony.

W połowie lat 90-tych Graddy Booch (Rational Software Corporation), Ivar Jacobson (Objectory) i James Rambaugh (General Electric) przystąpili do prac nad wzbogaceniem swoich metod elementami metod kolegów. Doprowadziło to do powstania zunifikowanego języka modelowania (Unified Modeling Language, UML). Robocza wersja UML 0.8 powstała w firmie Rational w październiku 1995. W 1996 roku powstało konsorcjum Object Management Group (OMG), które zrzeszało firmy zainteresowane rozwojem UML (m.in. DEC, Hewlett-Packard, IBM, Microsoft, Oracle, Rational czy IBM). W styczniu 1997 opublikowano specyfikacje UML 1.0, a w listopadzie 1997 UML 1.1.

W 2005 roku standardy UML 1.x zostały zastąpione przez UML 2.0, a następnie w 2009 roku przez UML 2.1.1 i 2.1.2 (oznaczane wspólnie jako UML 2.1). Obecnie najnowszą wersją języka UML jest UML 2.5 opracowany w 2012.

Narzędzia do modelowania UML

Narzędzia do modelowania UML są niezbędne. Wszystkie przedstawione narzędzia zgodne są ze specyfikacją UML 2.1, jednakże tylko niektóre obsługują UML 2.5.

Darmowe/otwarte:

- Astah
- Dia
- Eclipse
- StarUML
- Umbrello

Komercyjne:

- Astah
- Enterprise Architect
- IBM Rational Rose
- IBM Rational Software Architect
- Microsoft Visio
- Visual Paradigm for UML

Przegląd diagramów języka UML

Opis wybranych diagramów w UML 2.1 został zaczerpnięty z [1].

Diagram przypadków użycia

Diagram przypadków użycia (ang. use case diagram) jest diagramem, który przedstawia funkcjonalność systemu wraz z jego otoczeniem. Pozwala on na graficzne zaprezentowanie własności systemu tak, jak są one widziane po stronie użytkownika oraz usług jakie są widoczne z zewnątrz systemu.

Diagram pakietów

Diagram pakietów (ang. package diagram) jest strukturalnym diagramem, który prezentuje pakiety i relacje zachodzące pomiędzy nimi. Pozwala na modelowanie systemu na wysokim stopniu abstrakcji, gdyż pakiety reprezentują ogromną liczbę klas, interfejsów, diagramów i innych bytów, dzięki czemu możliwe jest wyeksponowanie tylko zasadniczych funkcji systemu.

Diagram klas

Diagram klas (ang. class diagram) obrazuje pewien zbiór klas, interfejsów i kooperacji oraz związki między nimi. Jest on grafem złożonym z wierzchołków (klas, interfejsów, kooperacji) i łuków (reprezentowanych przez relacje). Diagram stanowi statyczny opis systemu, który uwypukla związki pomiędzy klasami, pomijając pozostałe charakterystyki.

Diagram obiektów

Diagram obiektów (ang. object diagram) przedstawia obiekty (elementy modelu), związki między nimi (relacje: asocjacje, agregacje, kompozycje, generalizacje) oraz ograniczenia.

Diagram czynności

Diagram czynności (ang. activity diagram) jest diagramem interakcji, który służy do modelowania dynamicznych aspektów systemu. Jego zasadniczą funkcją jest przedstawienie sekwencji kroków, jakie są wykonane przez modelowany fragment systemu.

Diagram maszyny stanowej

Diagram maszyny stanowej przedstawia maszynę stanową, która zawiera proste stany i przejścia pomiędzy nimi.

Diagram sekwencji

Diagram sekwencji (ang. sequence diagram) służy do prezentowania interakcji pomiędzy obiektami wraz z uwzględnieniem komunikatów, jakie są przesyłane pomiędzy nimi w czasie. Zasadniczym zastosowaniem diagramów sekwencji jest modelowanie zachowania systemu w kontekście scenariuszy przypadków użycia.

Diagram komunikacji

Diagram komunikacji (communication diagram) był znany w UML 1.x pod nazwą diagramu współpracy (ang. collaboration diagram). Służy on do obrazowania, specyfikowania, tworzenia i dokumentowania dynamicznych aspektów ustalonego zestawu obiektów. Prezentuje organizację strukturalną obiektów wraz z komunikatami, które są wymieniane między nimi.

Diagram przebiegów czasowych

Diagram przebiegów czasowych (ang. timing diagram) to specjalistyczna forma diagramów sekwencji, która służy do prezentowania funkcjonowania obiektu, w sytuacji wysokich wymagań czasowych.

Diagram widoku interakcji

Diagram widoku interakcji (ang. interaction overview diagram) jest specjalną formą diagramu czynności i służy do prezentacji zależności i przepływu wiadomości pomiędzy interakcjami (diagramami interakcji).

Diagram komponentów

Diagram komponentów (ang. component diagram) służy do ilustracji organizacji i zależności pomiędzy komponentami. Diagram komponentów prezentuje system na wyższym poziomie abstrakcji niż diagram klas, gdyż każdy z komponentów może składać się z jednej lub większej liczby klas.

Diagram strukury

Diagram struktury (ang. composite structure diagram, internal structure diagram) służy do przedstawiania wewnętrznej struktury klasyfikatorów (tzn. klas, komponentów, węzłów, przypadków użycia) z uwzględnieniem punktów interakcji klasyfikatora z innymi częściami systemu.

Diagram wdrożenia

Diagram wdrożenia (ang. deployment diagram) służy do przedstawiania konfiguracji poszczególnych węzłów działającego systemu i zainstalowanych na nich komponentów. Węzłami systemu są najczęściej komputery i inny sprzęt komputerowy.

Notacja diagramów sekwencji UML 2.1

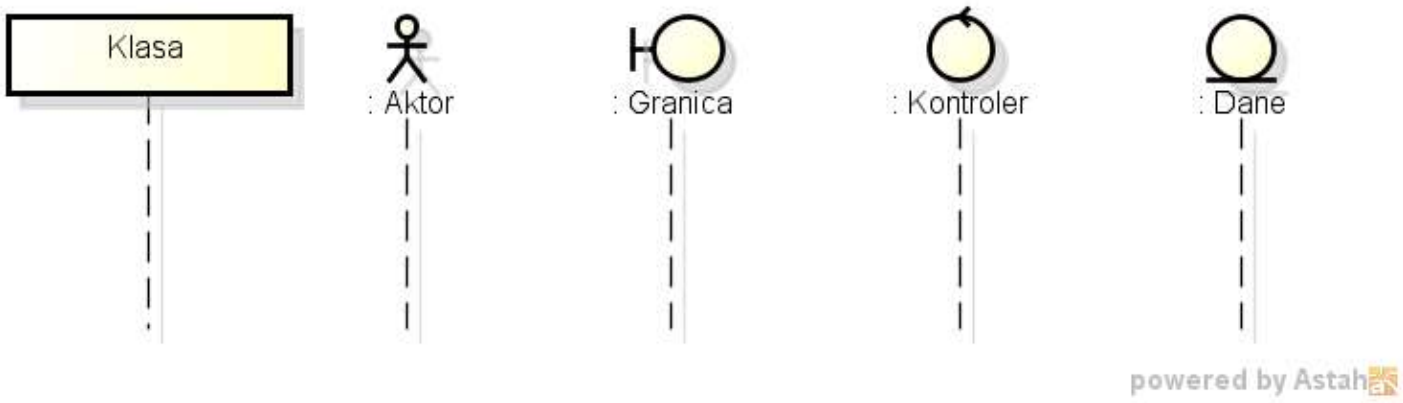
Linia życia

Znaczenie

Linia życia reprezentuje współuczestnika interakcji i czas jego istnienia podczas realizacji scenariusza. Linie życia najczęściej reprezentują konkretne obiekty (czasem klasy) i mogą przyjmować stereotypy, które świadczą o ich roli w systemie. Najczęściej spotykanymi stereotypami są:

- aktor (ang. actor)
- obiekt klasy granicznej (ang. boundary class)
- obiekt klasy sterującej (ang. controll class)
- obiekt klasy danych (ang. entity class)

Dla uproszczenia notacji każdy z wymienionych stereotypów posiadają własną reprezentację graficzną.



Implementacja

Byty reprezentowane za pomocą linii życia najczęściej implementowane są jako klasy programistyczne. Następująca linia życia zostanie zaimplementowana jako klasa:

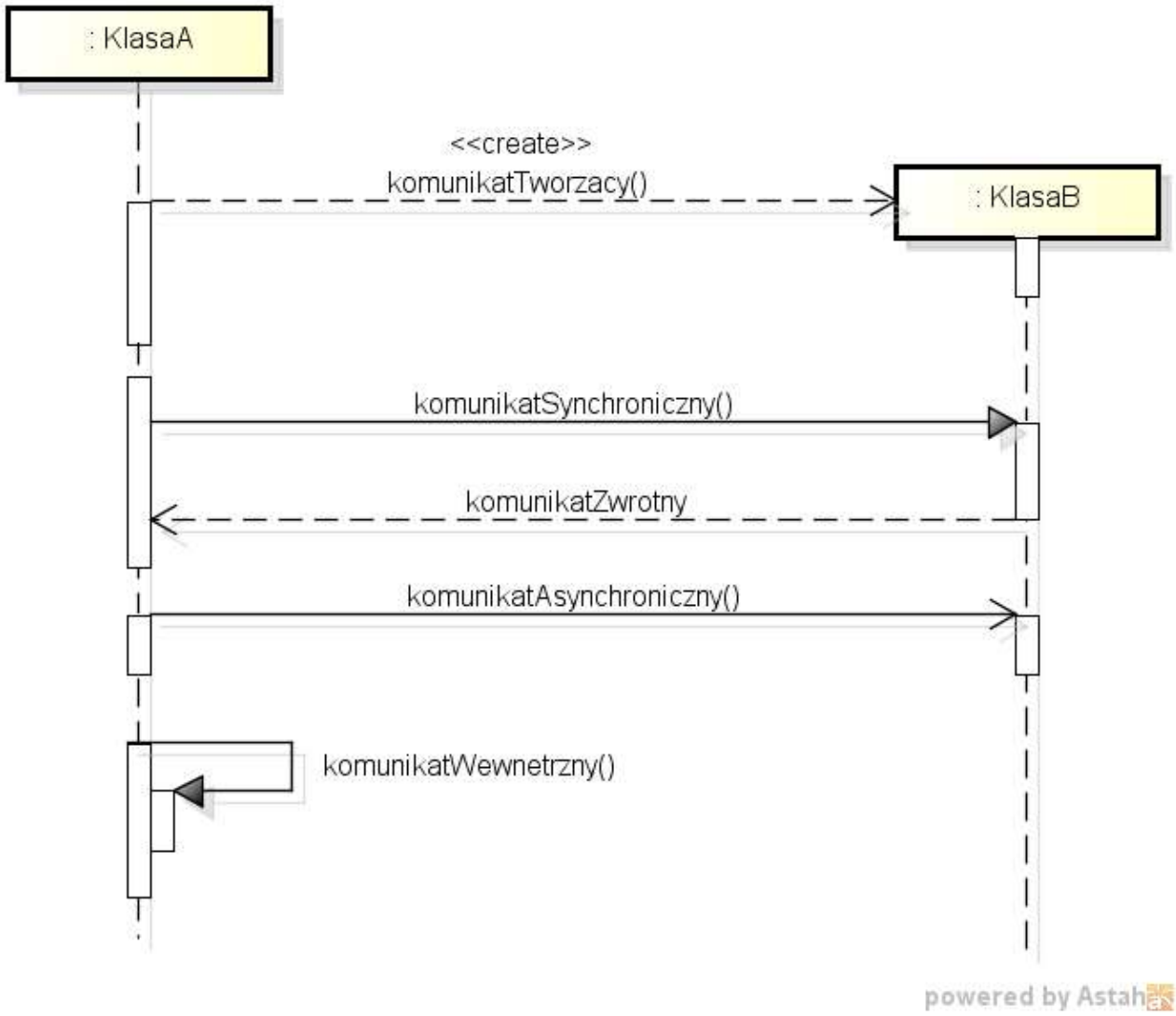
```
public class KlasaA{
   // ...
}
```

Komunikat

Znaczenie

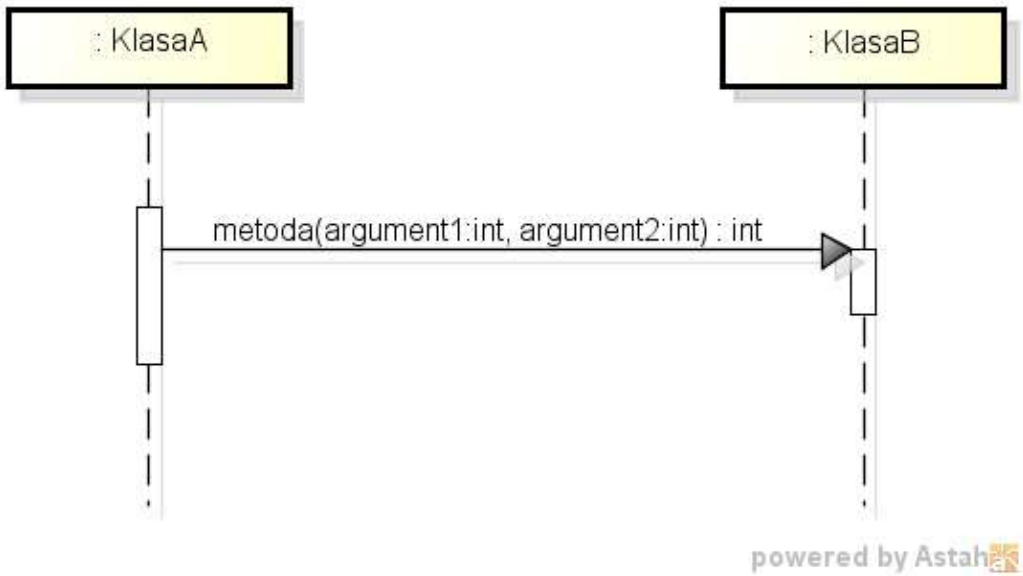
Komunikat (ang. message) jest to informacja przesyłana pomiędzy obiektami. Różne typy komunikatów zostały przedstawione na poniższym rysunku. Najczęściej wykorzystuje się komunikaty synchroniczne oznaczające, że natychmiast po jego wywołaniu przychodzi odpowiedź (wartość zwracana). Jeśli pożądane jest inne zachowanie należy wykorzystać komunikat zwrotny do jego opisu.

Innym typem wiadomości jest komunikat wewnętrzny, który obiekt wysyła do samego siebie. Ostatnim rodzajem komunikatów jest komunikat tworzący obiekt.



Implementacja

Komunikaty implementowane są za pomocą metod oraz ich wywołań.



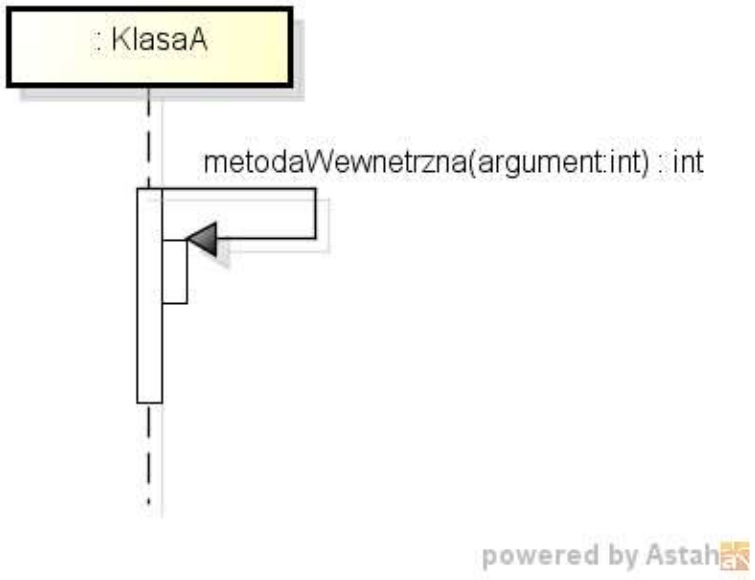

```
class KlasaA {
   // ...
   public void nazwa(){
      KlasaB obiektB;
      // ... (inicjalizacja zmiennej obiektB)
      int wartoscZwracana = obiektB.metoda(argument1, argument2);
      // ...
   }
```

```
   // ...
}

class KlasaB {
   // ...
   public int metoda(int argument1, int argument2){
      // ...
   }
}
```

Referencja do obiektu KlasaB jest jednak najczęściej przechowywana jako pole klasy KlasaA. W takiej sytuacji kod wygląda następująco:

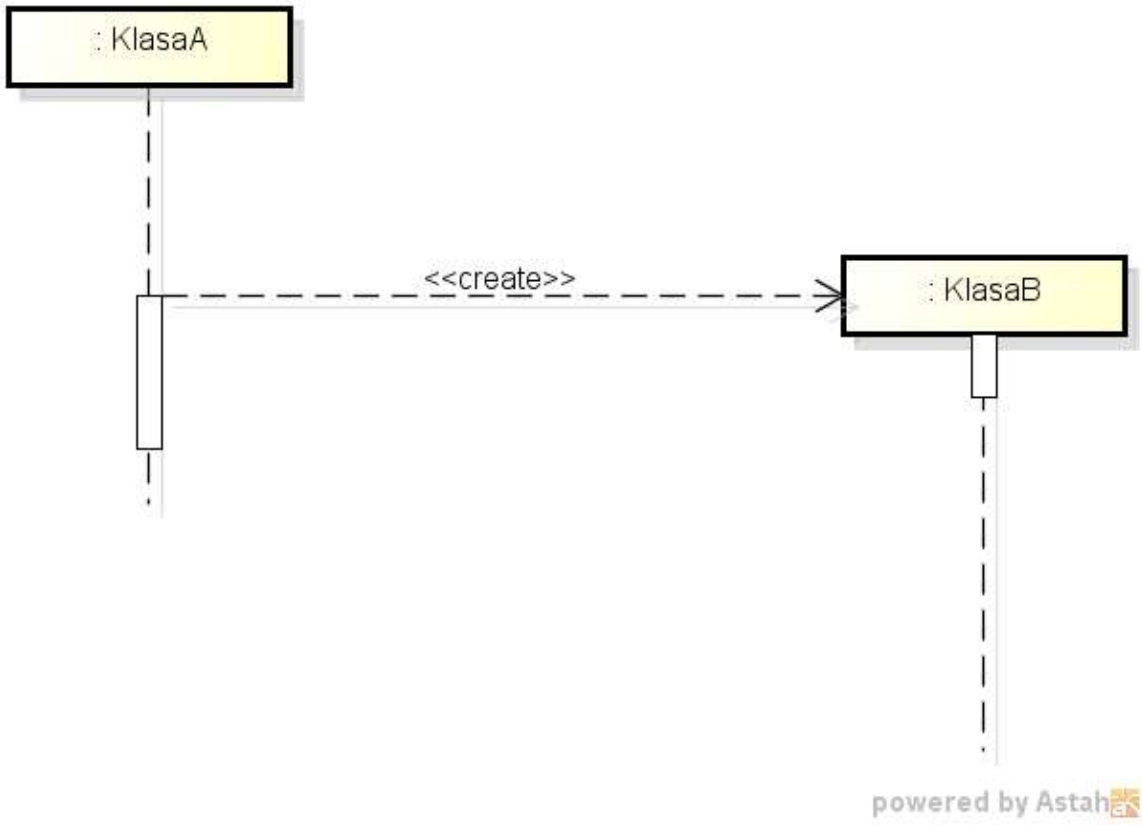
```
class KlasaA {
   private KlasaB obiektB;
   // ...
   public void nazwa(){
      // ... (inicjalizacja zmiennej obiektB)
      int wartoscZwracana = obiektB.metoda(argument1, argument2);
      // ...
   }
   // ...
}
```

Komunikat wewnętrzny realizowany jest najczęściej jako wywołanie metody prywatnej:




```
class KlasaA {
   // ...
   public void metoda(){
      // ...
      int wartoscZwracana = metodaWewnetrzna(argument);
      // ...
   }

   private int metodaWewnetrzna(int argument) {
       // ...
   }
   // ...
}
```

Komunikaty tworzące obiekty implementowane są najczęściej za pomocą wywołania konstruktora:




```
class KlasaA {
   // ...
   public void nazwa(){
      KlasaB obiektB = new KlasaB(argument); // wywołanie konstruktora
      // ...
   }
   // ...
}

class KlasaB {
   // ...
   public KlasaB(int argument){ // konstruktor klasy KlasaB
      // ...
   }
}
```

Fragment

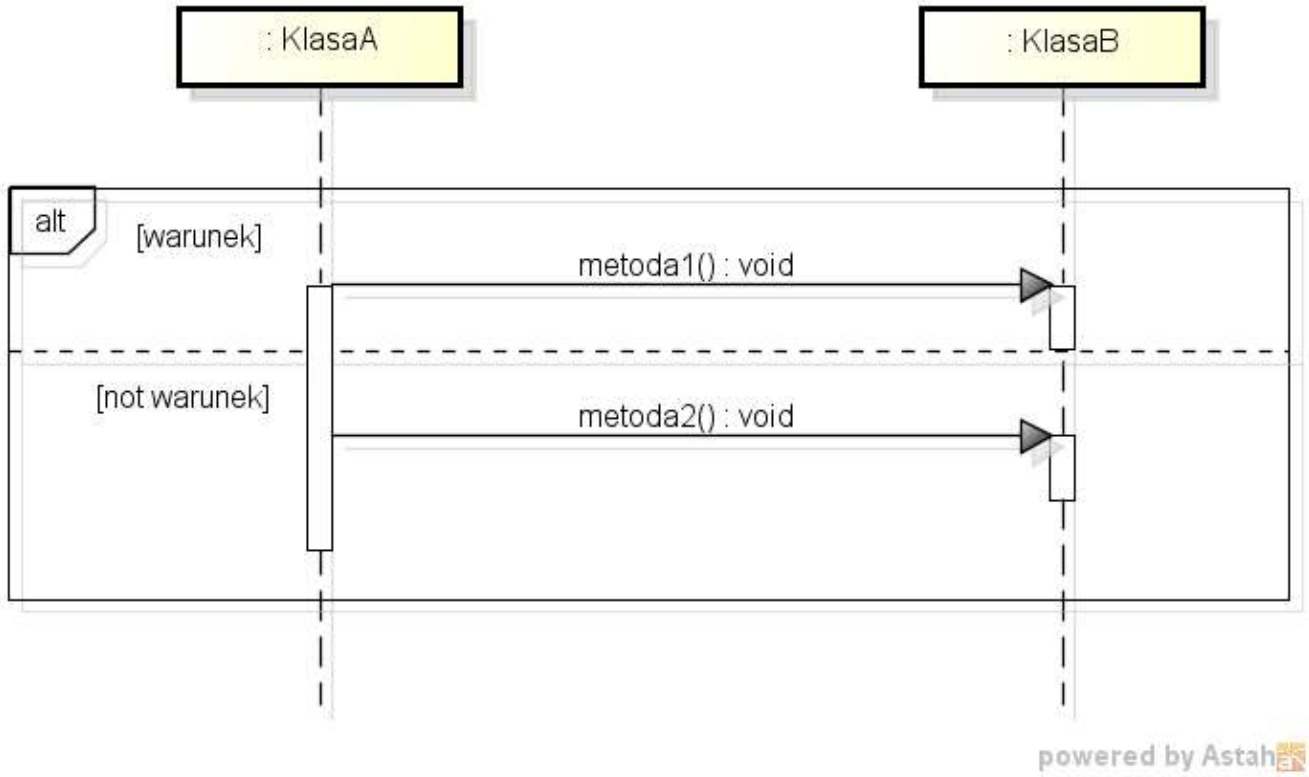
Znaczenie

Fragment (ang. combined fragment) to konceptualnie zamknięta część diagramu sekwencji, która rozszerza możliwości obejmowanego przez siebie obszaru diagramu sekwencji. Fragment może reprezentować pętle, powtórzenia, czy scenariusze alternatywne. Rodzaj fragmentu jest określany przez umieszczenie odpowiedniego słowa kluczowego w lewym górnym rogu. Najczęściej stosowane są fragmenty typu:

- **alt -** pozwala na podzielenie diagramu sekwencji na kilka alternatywnych scenariuszy. To który z nich będzie realizowany zależy od spełnienia podanych warunków.
- **break -** pozwala na zakończenie scenariusza w dowolnym momencie. Fragment zawiera kroki jakie zostaną wywołane tuż przez wyjściem ze scenariusza.
- **loop -** powtarzanie danego fragmentu określoną liczbę razy lub dopóki spełniony jest zadany warunek.
- **neg -** pozwala na przedstawienie niepoprawnej sekwencji kroków.
- **par -** wysyłanie komunikatów w takim fragmencie odbywa się równolegle.

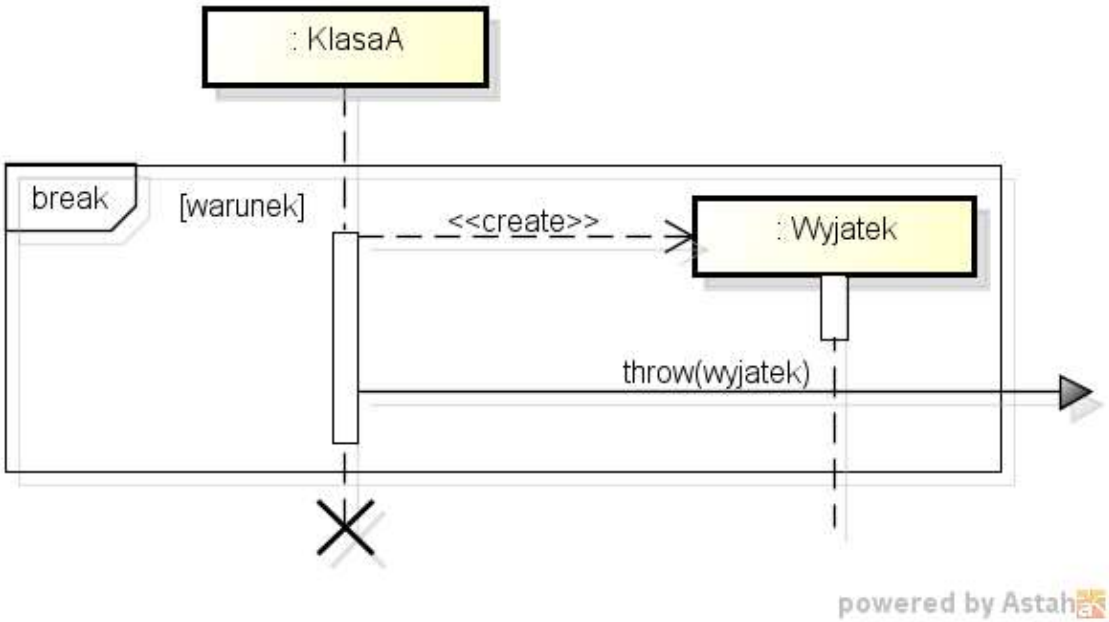
Implementacja

Fragment typu **alt** najczęściej realizowany jest za pomocą instrukcji warunkowej.



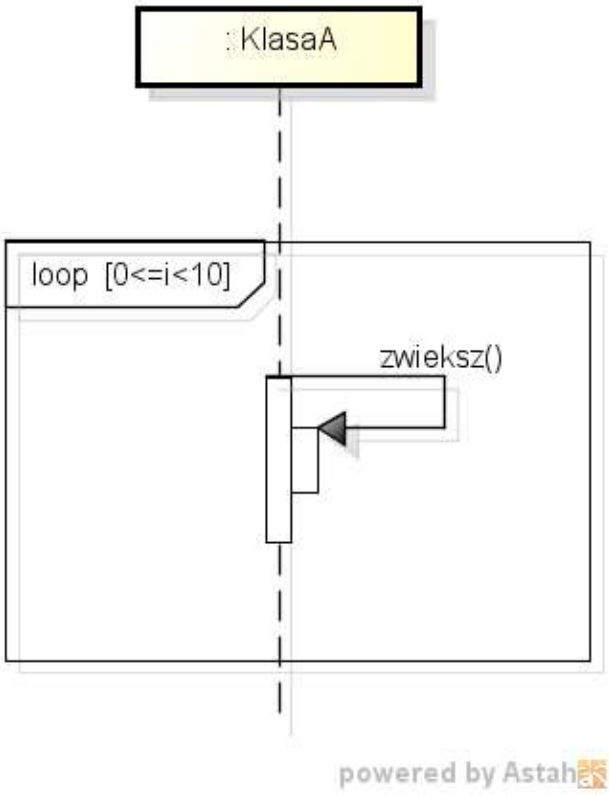

```
class KlasaA {
   // ...
   public void nazwa() {
      // ...
      KlasaB obiektB;
      // ... (inicjalizacja zmiennej obiektB)
      if(warunek) {
         obiektB.metoda1();
      } else {
         obiektB.metoda2();
      }
      // ...
   }
   // ...
}

class KlasaB {
   // ...
   public void metoda1(){
      // ...
   }
   public void metoda2(){

     // ...
   }
}
```

Fragmenty typu **break** modelujące sytuacje awaryjne najczęściej implementowane są za pomocą wyjątków.




```
class KlasaA {
   // ...
   public void nazwa() {
      // ...
      if(warunek) {
         throw new Wyjatek();
      }
      // ...
   }
   // ...
}

class Wyjatek extends Exception {
   // ...
}
```

Fragmenty typu **loop** najczęściej implementuje się z wykorzystaniem pętli for lub while.




```
class KlasaA {
   // ...
   private licznik = 0;
```

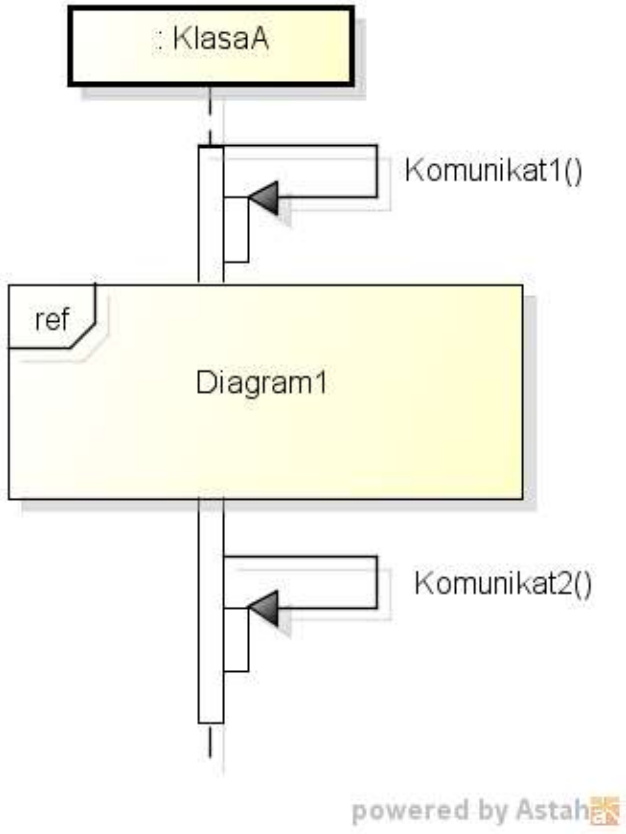
```
   public void nazwa() {
      // ...
      for(int i=0;i<10;i++) {
         zwieksz();
      }
      // ...
   }

   private void zwieksz() {
      licznik++;
   }
   // ...
}
```

Fragmenty typu **neg** nie powinny być implementowane, gdyż przedstawiają niepoprawne sekwencje komunikatów. Takie fragmenty mogą posłużyć do implementacji mechanizmu sprawdzania poprawności stanu aplikacji. Fragmenty typu **par** implementowane są najczęściej przy użyciu wątków lub osobnych procesów.

Wystąpienie interakcji

Znaczenie

Wystąpienie interakcji (ang. interaction occurence) jest odniesieniem do interakcji, której obraz przedstawiany jest na innym diagramie. Użycie referencji powoduje, że diagram staje się bardziej czytelny, gdyż pozwala na ukrycie szczegółów nieistotnych z punktu widzenia modelowanej sytuacji. Dodatkowo zastosowanie referencji do innego diagramu umożliwia wykorzystanie tych samych elementów w wielu miejscach projektu.



Implementacja

Wystąpienie interakcji implementowane jest tak samo jak wysłanie komunikatu.

Notacja diagramów komunikacji UML 2.1

Obiekt

Znaczenie

Obiekt jest egzemplarzem klasy. Obiekty w diagramach komunikacji mogą przyjmować stereotypy, które świadczą o roli, jaką pełni dany obiekt w systemie. Najczęściej spotykanymi stereotypami są:

- aktor (ang. actor)
- obiekt klasy granicznej (ang. boundary class)
- obiekt klasy sterującej (ang. controll class)
- obiekt klasy danych (ang. entity class)

Dla uproszczenia notacji każdy z wymienionych stereotypów posiadają własną reprezentację graficzną.

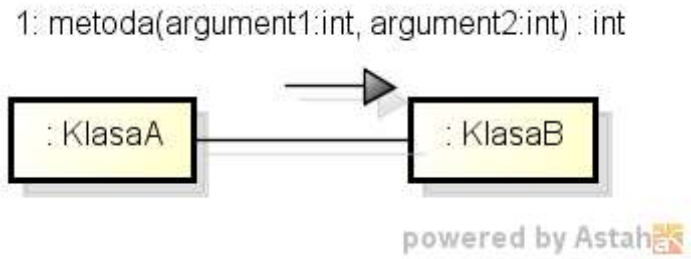
Implementacja

Implementacja obiektów z diagramów komunikacji polega najczęściej na pozyskiwaniu i przechowywaniu referencji do obiektu konkretnej klasy. Więcej szczegółów podane jest w materiałach o liniach życia w diagramach sekwencji.

Komunikat

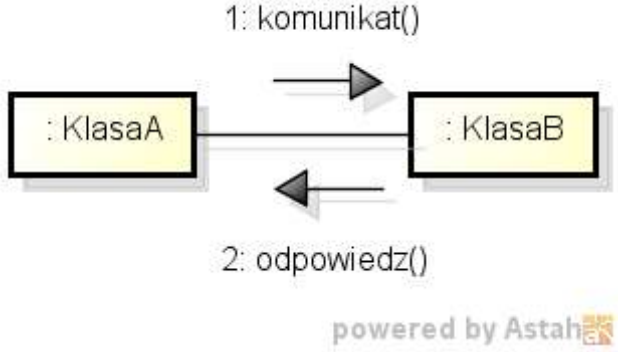
Znaczenie

Komunikat (ang. message) jest wiązaniem łączącym obiekty. Na diagramach komunikacji obiekty poza połączeniami między obiektami posiadają ukierunkowane strzałką z nazwą komunikatów, które przechodzą pomiędzy obiektami. Komunikaty są numerowane, co pozwala stwierdzić w jakiej kolejności się ze sobą komunikują.



Implementacja

Implementacja komunikatów jest analogiczna jak w przypadku diagramów sekwencji. Należy pamiętać, że obiekt wysyłający komunikat musi posiadać referencje do obiektu odbierającego. Częstym błędem jest modelowanie odpowiedzi obiektu na komunikat również za pomocą komunikatu:



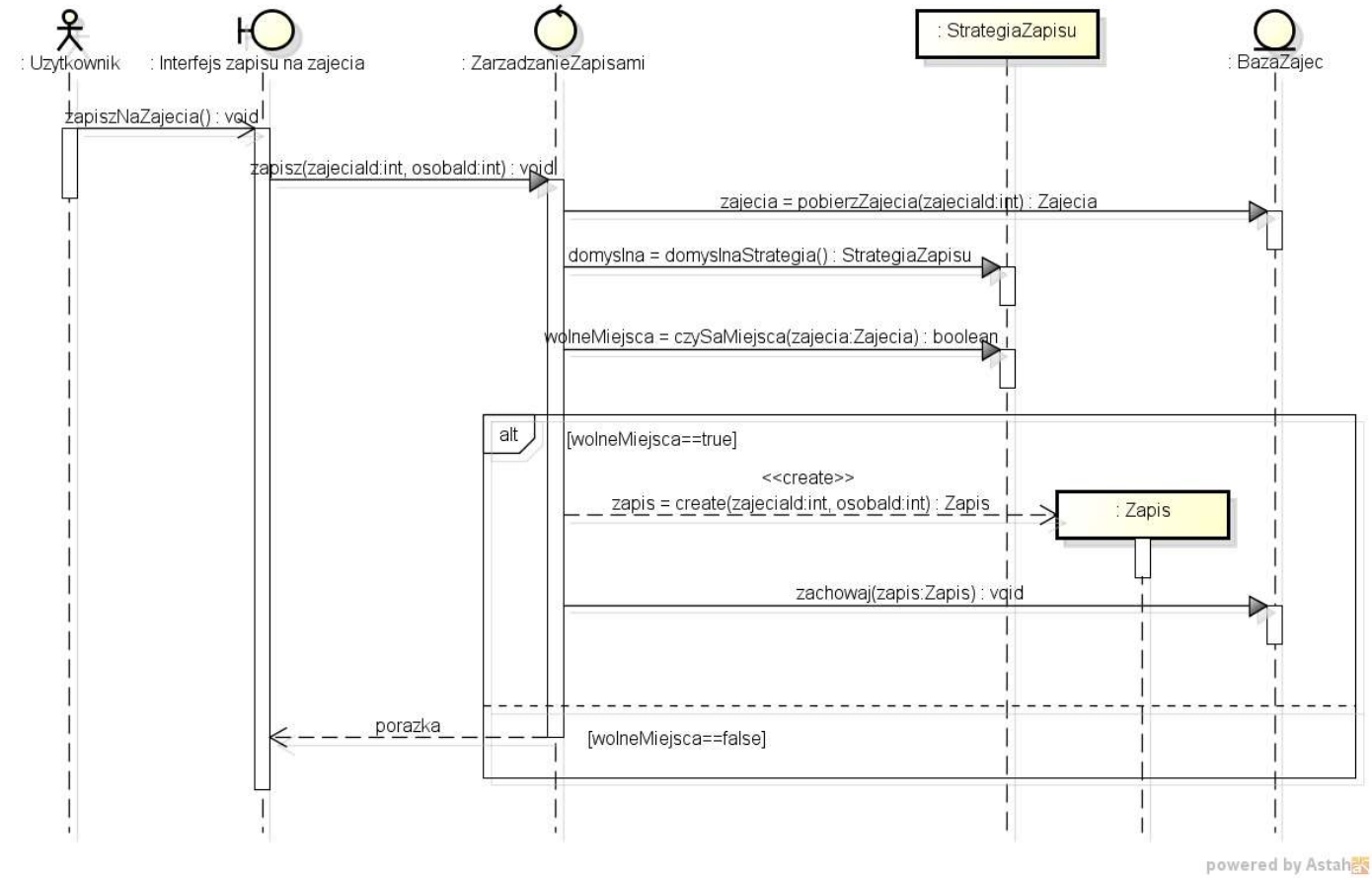
Taki zapis oznacza, że oba obiekty mają wzajemnie referencje do siebie co często nie jest celem osoby tworzącej diagram.

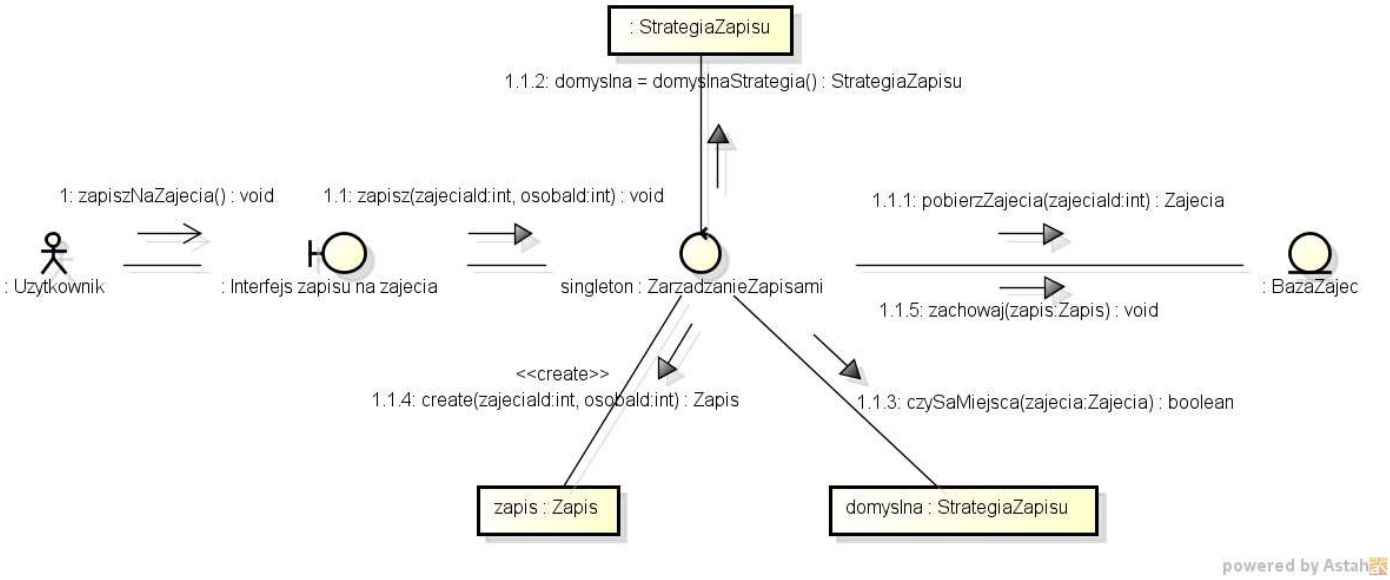
```
class KlasaA {

   private KlasaB obiektB;

   // ...

}

class KlasaB {

   private KlasaA obiektA;

   // ...

}
```

Oczywiście taka sytuacja w pewnych systemach może być w pełni poprawna dlatego należy zachować szczególną ostrożność.

Przykład programistyczny

Poniżej przedstawiono dwa diagramy interakcji (sekwencji i komunikacji), które przedstawiają proces zapisu na zajęcia. Oba typy diagramów potrafią prezentować bardzo zbliżone zachowania systemu. Wybór odpowiedniego diagramu zależy od konkretnego systemu i modelowanego zachowania.





Poniżej znajduje się przykładowa implementacja zachowania przedstawionego na obu diagramach interakcji. Inicjująca akcja użytkownika zostanie zasymulowana przez wywołanie metody main klasy Main.

```
public class Main {

   public static void main(String[] agrs) {

      InterfejsZapisuNaZajecia interfejs = new InterfejsZapisuNaZajecia();

      interfejs.zapiszNaZajecia();

   }

}
```

InterfejsZapisuNaZajecia jest graficzny, po otwarciu okna użytkownik wybiera grupę zajęciową oraz podaje swoje ID. Gdy użytkownik dokona wyboru wywoływana jest metoda kontrolera, która realizuje logikę tego procesu.

```
class InterfejsZapisuNaZajecia {

   private ZarzadzanieZapisami kontroler = new ZarzadzanieZapisami();

   public void zapiszNaZajecia() {

      // oczekiwanie aż użytkownik wybierze grupę zajęciową oraz wprowadzi swoje ID

      boolean sukces = kontroler.zapisz(pobierzZajeciaId(), pobierzOsobaId());
      if (sukces) {
         // powiadom użytkownika o sukcesie
      } else {
         // powiadom uzytkownika o porażce
      }

   }

   private int pobierzOsobaId() {

   // ...

   }

   private int pobierzZajeciaId() {

   // ...

   }

}

class ZarzadzanieZapisami {

   private BazaZajec baza = new BazaZajec();

   public boolean zapisz(int zajeciaId, int osobaId) {

      Zajecia zajecia = baza.pobierzZacjecia(zajeciaId);

      StrategiaZapisu domyslna = StrategiaZapisu.domyslna();

      boolean wolneMiejsca = domyslna.czySaWolneMiejsca(zajecia);

      if (wolneMiejsca) {

         Zapis zapis = new Zapis(zajeciaId, osobaId);

         baza.zachowaj(zapis);

         return true

      } else {

         return false;

      }

   }

}

class StrategiaZapisu {

   private StrategiaZapisu domyslna = new StrategiaZapisu();

   private StrategiaZapisu() {

   }

   public static StrategiaZapisu domyslna() {

      return domyslna;

   }

   public boolean czySaWolneMiejsca(Zajecia zajecia) {

      return zajecia.ileZapisanych() < zajecia.limitMiejsc();

   }

}

class Zajecia {

   private int zapisanych, limitMiejsc, id;
```

```
   public Zajecia(int id, int zapisanych, int limitMiejsc) {
      this.id = id;

      this.zapisanych = zapisanych;

      this.limitMiejsc = limitMiejsc;

   }

   public int ileZapisanych() {

      return zapisanych;

   }
   public void dodaj() {
      zapisanych++;
   }

   public int limitMiejsc() {

      return limitMiejsc;

   }
   public int id() {
      return id;
   }

}

class BazaZajec {
   // baza zajec symulowana przez mapę przygotowanych wartości
   private Map<Integer, Zajecia> zajecia = new HashMap<>();
   public BazaZajec() {
      zajecia.put(1, new Zajecia(1, 3, 15);
      zajecia.put(2, new Zajecia(2, 7, 24);
      zajecia.put(3, new Zajecia(3, 0, 100);
   }

   public Zajecia pobierzZajecia(int zajeciaId) {
      return zajecia.get(zajeciaId);
   }
   public void zachowaj(Zapis zapis) {
      Zajecia z = zajecia.get(zapis.zajeciaId());
      z.dodaj();
   }

}
class Zapis {
   int zajeciaId, osobaId;
   public Zapis(int zajeciaId, int osobaId) {
      this.zajeciaId = zajeciaId;
      this.osobaId = osobaId;
   }
   public int zajeciaId() {
      return zajeciaId;
   }
}
```

Wzorzec projektowy (ang. *design pattern*) – w inżynierii oprogramowania, uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się, powtarzalnych problemów projektowych. Pokazuje powiązania i zależności pomiędzy klasami oraz obiektami i ułatwia tworzenie, modyfikację oraz pielęgnację kodu źródłowego. Jest opisem rozwiązania, a nie jego implementacją. Wzorce projektowe stosowane są w projektach wykorzystujących programowanie obiektowe.

Wzorce projektowe w informatyce wywodzą się z wzorców projektowych w architekturze, które zostały zaproponowane przez amerykańskiego architekta Christophera Alexandra i miały ułatwić konstruowanie mieszkań i pomieszczeń biurowych. Pomysł ten nie został jednak przyjęty.

Inaczej stało się w informatyce. Termin wzorca projektowego został wprowadzony do inżynierii oprogramowania przez Kenta Becka oraz Warda Cunninghama w 1987 roku. Na konferencji OOPSLA, przedstawili oni wyniki swojego eksperymentu dotyczącego ich zastosowania w programowaniu. Został spopularyzowany w 1995 przez tzw. *Bandę Czterech* (Erich Gamma, Richard Helm, Ralph Johnson oraz John Vlissides) dzięki książce *Inżynieria oprogramowania: Wzorce projektowe*.

Wzorce projektowe najczęściej tworzone są w oparciu o programowanie obiektowe. Zakres tego pojęcia stał się problemem rozważanym od połowy lat 90. XX wieku. Ostatecznie ustalono, że algorytmy nie są wzorcami projektowymi, jako że rozwiązują problemy obliczeniowe, a nie projektowe. Wzorce często są łączone w celu rozwiązania bardziej złożonego problemu.

Zamiast skupiać się na funkcjonowaniu poszczególnych elementów, wzorce projektowe stanowią abstrakcyjny opis zależności pomiędzy klasami, co w efekcie wprowadza pewną standaryzację kodu oraz zwiększa jego zrozumiałość, wydajność i niezawodność. Wartość wzorców projektowych stanowi nie tylko samo rozwiązanie problemu, ale także dokumentacja, która wyjaśnia cel, działanie, zalety danego rozwiązania, co pomaga w łatwiejszym stosowaniu i adaptacji wzorców w danym zastosowaniu.

Wzorce projektowe mogą przyspieszyć proces rozwoju oprogramowania przez dostarczenie wypróbowanych rozwiązań dla problemów, które mogą nie być oczywiste na początku procesu projektowego. Często zagadnienia te wiążą się z ewolucją oczekiwań względem projektowanego systemu: rozszerzeniem jego funkcjonalności, zmianą sposobu i formatu wprowadzanych danych czy dostosowaniem aplikacji do różnych klas użytkowników. Nieuwzględnienie ich na początku procesu rozwoju produktu programistycznego powoduje często konieczność gruntownego przebudowywania zaawansowanego lub gotowego już oprogramowania.

Źródło: Wikipedia

Tworzenie obiektów

Jednym z najprostszych wzorców projektowych jest wzorzec Singleton. Istnieje bardzo wiele jego modyfikacji oraz różnych implementacji. Poniżej przedstawione zostaną najczęściej wykorzystywane wraz ze wzorcami towarzyszącymi.

Podstawowym założeniem wzorca singleton jest stworzenie obiektu, który będzie posiadał tylko jedną instancję. Takie ograniczenie jest często niezbędne dla poprawnego działania aplikacji. Najczęściej podawanym przykładem jest połączenie z bazą danych. Wielokrotne połączenie z tą samą bazą mogło by prowadzić do problemów z synchronizacją danych i pamięcią podręczną. Dla ilustracji przykładu posłuży nam następująca bardzo prosta klasa:

```
class PolaczenieZBazaDanych {

   Dane pobierzDane(){

      // implementacja pobierania danych

   }

}
```

Chcemy aby w systemie istniała tylko jedna instancja tej klasy. Najprostszym rozwiązaniem jest uniemożliwienie wywołania konstruktora wszędzie poza jednym miejscem w aplikacji.

```
class PolaczenieZBazaDanych {
   private static PolaczenieZBazaDanych domyslne; // statyczne pole klasy przechowuje jedyną instancję

   private PolaczenieZBazaDanych() { // prywatny konstruktor nie jest dostępny poza dane klasa
   }

   public static PolaczenieZBazaDanych domyslne() { // publiczna metoda umożliwiająca pobranie instancji
      if(domyslne == null) {                        // konstruktor zostanie wywołany tylko raz, przy pierwszym wywołaniu metody
         domyslne = new PolaczenieZBazaDanych();
      }
      return domyslne;
   }

   Dane pobierzDane(){

      // implementacja pobierania danych

   }

}
```

Szczególną uwagę należy zwrócić na 2 elementy

- prywatny konstruktor uniemożliwia stworzenie obiektów tej klasy z zewnątrz
- statyczna metoda tworząca i udostępniająca referencje do jedynego obiektu tej klasy

Wiele współczesnych języków programowania wyposażonych jest w mechanizm refleksji, który pozwala na modyfikowanie kodu aplikacji w trakcie jej działania. Uniemożliwia to implementacje wzorca singleton w 100% odporną na manipulację. Nie powinno być to jednak celem, gdyż kod aplikacji (a szczególnie wzorce projektowe) mają być przede wszystkim czytelne dla programisty. Brak publicznego konstruktora ma jednoznaczne znaczenie - należy poszukać innego sposoby na zdobycie obiektu danego typu.

Powyższy kod jest również przykładem wykorzystania innego wzorca projektowego - metody wytwórczej. Metoda domyslne() tworzy obiekt typu PolaczenieZBazaDanych zastępując w ten sposób konstruktor. Takie rozwiązanie jest często preferowane, gdyż bezpośrednie wywoływanie konstruktora na stałe wiąże kod z konkretną implementacją. W przypadku chęci zmiany implementacji (np. na podklasę obecnego typu) wszędzie konieczna jest zmiana wywołań starego konstruktora na nowy. W przypadku zastosowania metody wytwórczej wystarczy podmienić konstruktor w jednym miejscu.

Załóżmy ze w systemie pojawia się druga baza danych, jej implementacja również oparta jest o wzorce singleton i metoda wytwórcza:

```
class PolaczenieZDrugaBazaDanych {
   private static PolaczenieZBazaDanych domyslne; // statyczne pole klasy przechowuje jedyną instancję

   private PolaczenieZBazaDanych() { // prywatny konstruktor nie jest dostępny poza dane klasa
   }

   public static PolaczenieZDrugaBazaDanych domyslne() { // publiczna metoda umożliwiająca pobranie instancji
      if(domyslne == null) {                             // konstruktor zostanie wywołany tylko raz, przy pierwszym wywołaniu metody
         domyslne = new PolaczenieZBazaDanych();
      }
      return domyslne;
   }

   Dane pobierzDane(){

      // implementacja pobierania danych

   }

}
```

W takiej sytuacji za każdym razem chcąc pobrać dane z bazy musimy wiedzieć, którą z metod wykonać: PolaczenieZBazaDanych.domyslne() czy PolaczenieZDrugaBazaDanych.domyslne(). Co jeśli system obsługuje wiele typów bazy danych jednak w danej chwili wykorzystywana jest tylko jedna konkretna implementacja. Przy każdej interakcji z bazą danych konieczne by było sprawdzanie której bazy użyć:

```
public class Main {

   public static void main(String[] args) {

      // ...

      if (Konfiguracja.bazaPierwsza()) {

         Dane dane = PolaczenieZPierwszaBaza.domyslne().pobierzDane();

      } else {

         if (Konfiguracja.bazaDruga()) {

            Dane dane = PolaczenieZDrugaBaza.domyslne().pobierzDane();

         }
      }

   }

}

class Konfiguracja {

   static boolean bazaPierwsza(){

      return true; // w zależności od konfiguracji

   }

   static boolean bazaDruga(){

      return false; // w zależności od konfiguracji

   }

}
```

Przy większej ilości baz takie rozwiązanie przestaje być efektywne. Znacznie lepiej było by przenieść odpowiedzialność za wybór bazy danych do osobnego obiektu. Z pomocą przychodzi wzorzec fabryka. Aby móc go zastosować w tej sytuacji konieczne jest dodanie interfejsu, który będzie implementowany przez wszystkie połączenia z bazą danych.

```
interface PolaczenieZBaza {
   Dane pobierzDane();
}

class PolaczenieZPierwszaBazaDanych implements PolaczenieZBaza {
   // ...
}

class PolaczenieZDrugaBazaDanych implements PolaczenieZBaza {
   // ...
}
```

Teraz można przejść do implementacji fabryki, która również najczęściej jest singletonem.

```
class FabrykaPolaczen {
   private static FabrykaPolaczen domyslna = new FabrykaPolaczen();

   public static FabrykaPolaczen domyslna() {
      return domyslna;
   }

   private FabrykaPolaczen(){
   }

   public PolaczenieZBaza pobierzPolaczenie() {

      if (Konfiguracja.bazaPierwsza()) {

         return PolaczenieZPierwszaBaza.domyslne();

      } else {

         if (Konfiguracja.bazaDruga()) {

            Dane dane = PolaczenieZDrugaBaza.domyslne();

         }
      }

   }
```

```
}

public class Main {

   public static void main(String[] args) {

      // ...

      PolaczenieZBaza polaczenie = FabrykaPolaczen.domyslna().pobierzPolaczenie();
      Dane dane = polaczenie.pobierzDane();

   }

}
```

Jak widać interakcja z baza danych jest teraz znacznie prostsza. Nie ma potrzeby sprawdzania jakiej bazy użyć w danym momencie, o wszystkim decyduje fabryka. Należy zwrócić uwagę, jest to stosunkowo nietypowe zastosowanie wzorca fabryka, gdyż tak właściwie nie tworzy ona żadnych obiektów. Spowodowane jest to tym, że obiekty są singletonami tworzonymi w metodach wytwórczych.

W dużych systemach informatycznych liczba fabryk i metod wytwórczych może być bardzo duża. Ciągłe ich stosowanie, a przede wszystkim implementowanie bardzo podobnej logiki w bardzo wielu różnych klasach doprowadziło do powstania zupełnie nowej koncepcji zarządzania tworzeniem obiektów. Wstrzykiwanie zależności (ang. Dependency Injection, DI) jest odpowiedzią na ten problem. Zamiast tworzyć fabryki i metody wytwórcze odpowiednie klasy wzbogacane są o metadane (anotacje), które umożliwiają zewnętrznemu oprogramowaniu (najczęściej jest to specjalna biblioteka korzystająca z refleksji lub natywne oprogramowanie zintegrowane ze środowiskiem wykonawczym) dostarczenie (wstrzyknięcie) referencji do odpowiednich obiektów. Powyższy przykład zapisany z wykorzystaniem takiego mechanizmu wyglądał by następująco:

```
@Zasob(singleton = true, domyslny = true)
class PolaczenieZPierwszaBazaDanych {

   Dane pobierzDane(){

      // implementacja pobierania danych

   }

}


@Zasob(singleton = true, domyslny = false)
class PolaczenieZDrugaBazaDanych {

   Dane pobierzDane(){

      // implementacja pobierania danych

   }

}

public class Main {
   @Inject
   private PolaczenieZBaza polaczenie;

   public static void main(String[] args) {

      // ...
      Dane dane = polaczenie.pobierzDane();

   }
}
```

W klasie Main pole polaczenie oznaczone jest anotacja @Inject. Zewnętrzna biblioteka podczas tworzenia obiektu tej klasy szuka klas implementujących interfejs PolaczenieZBaza (zgodnie z typem pola), które oznaczone są anotacją @Zasob. W tym przykładzie dostępne są dwa takie obiekty, jeden oznaczony jest jako domyślny i to on zostanie stworzony i wstrzyknięty. Anotacja @Zasob umożliwia oznaczenie klasy jako singleton, dzięki czemu biblioteka zadba aby w systemie utworzona została tylko jedna jej instancja.

Przykładami bibliotek, które implementują wstrzykiwanie zależności są:

- Spring Framework - Java
- kontener EJB - Java Enterprise Edition
- Guice - Java
- GIN - GWT
- MEF - .NET
- AngularJS - JavaScript

Przedstawiona w przykładzie notacja, jest znacznie uproszczona. Większość bibliotek wymaga specyficznej konfiguracji oraz spełnienia odpowiednich warunków.

Synchronizacja stanu obiektów

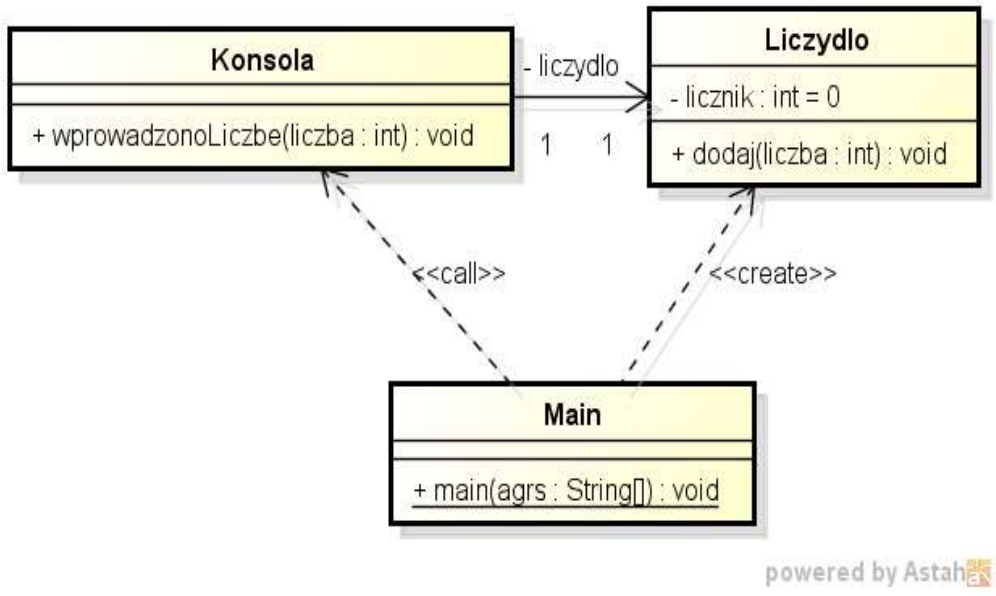
We współczesnych systemach często zachodzi potrzeba utrzymania synchronizacji stanu pomiędzy różnymi obiektami. Przykładowo wyobraźmy sobie sytuację w której program oczekuje na wpisanie przez użytkownika wartości liczbowej aby dodać ją do wcześniej zdefiniowanego licznika. W poniższym przykładzie mamy dwie klasy, dla których chcemy synchronizować stan ich instancji.

```
class Konsola {

   void wprowadzonoLiczbe(int liczba){

      // ...

   }

}

class Liczydlo {

   private int licznik = 0;

   public void dodaj(int liczba){

      licznik += liczba;

   }

}

// ...
public static void main(String[] args){
   Konsola konsola = new Konsola();

   konsola.wprowadzonoLiczbe(5);
}
// ...
```

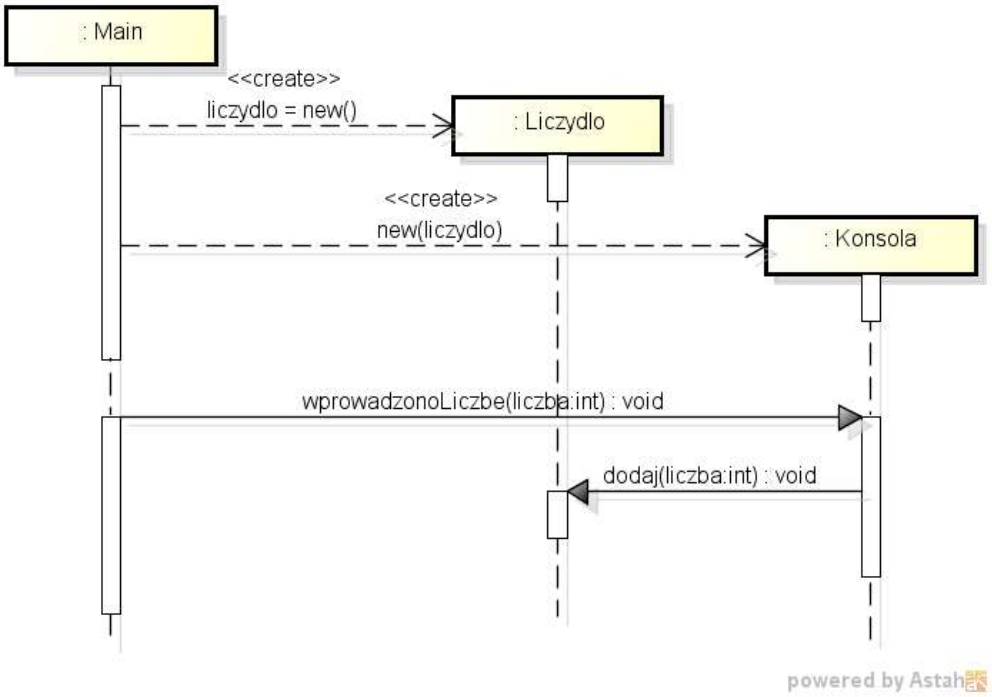
Klasa Konsola reprezentuje interfejs użytkownika. Chcemy aby każde wywołanie metody wprowadzonoLiczbe (użytkownik wpisał liczbę w konsoli) powodowało dodanie jej do wcześniej zdefiniowanego licznika (reprezentowanego przez klasę Liczydło). Najprostsze i bezpośrednie rozwiązanie polega na dodaniu wywołania metody dodaj() klasy Liczydlo wewnątrz metody wprowadzonoLiczbe().

```
class Konsola {
   private Liczydło liczydlo;
   Konsola(Liczydlo liczydlo){
      this.liczydlo = liczydlo;
   }

   void wprowadzonoLiczbe(int liczba){

      liczydlo.dodaj(liczba);

   }

}

// ...
public static void main(String[] args){
   Liczydlo liczydlo = new Liczydlo();
   Konsola konsola = new Konsola(liczydlo);

   konsola.wprowadzonoLiczbe(5);
}
// ...
```

Poniższy diagram klas przedstawia to rozwiązanie.



Takie rozwiązanie jest bardzo proste i przejrzyste. Obiekt przekazuje informacje o zmianie swojego stanu bezpośrednio do zainteresowanego za pomocą wywołania metody. Zachowanie to przedstawione na diagramie sekwencji wygląda następująco



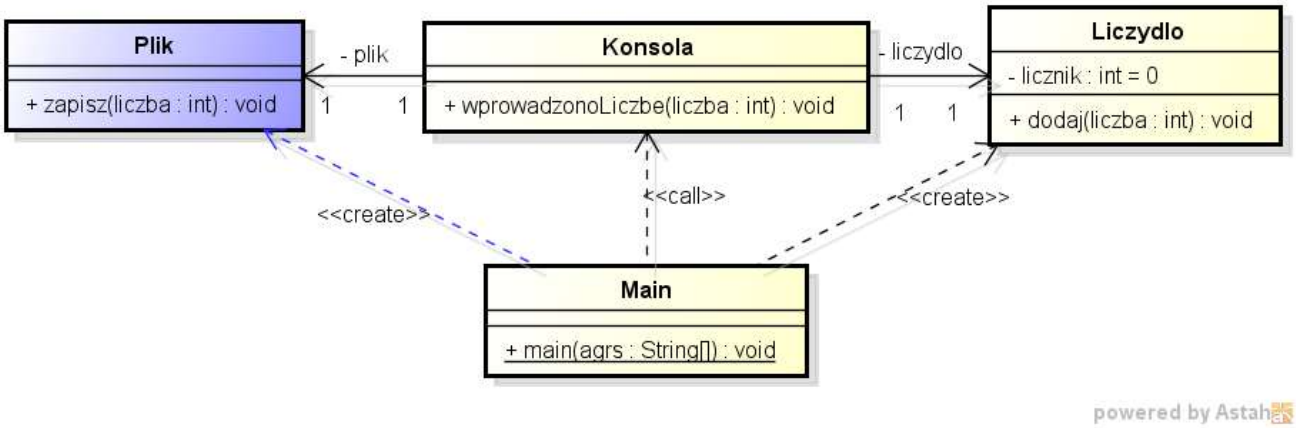
Niestety nawet w prostych systemach może okazać się bardzo kłopotliwe. W miarę rozwoju programu pojawiła się potrzeba zapisania historii wszystkich wprowadzonych liczb. Do zapisu wykorzystana zostanie następująca klasa:

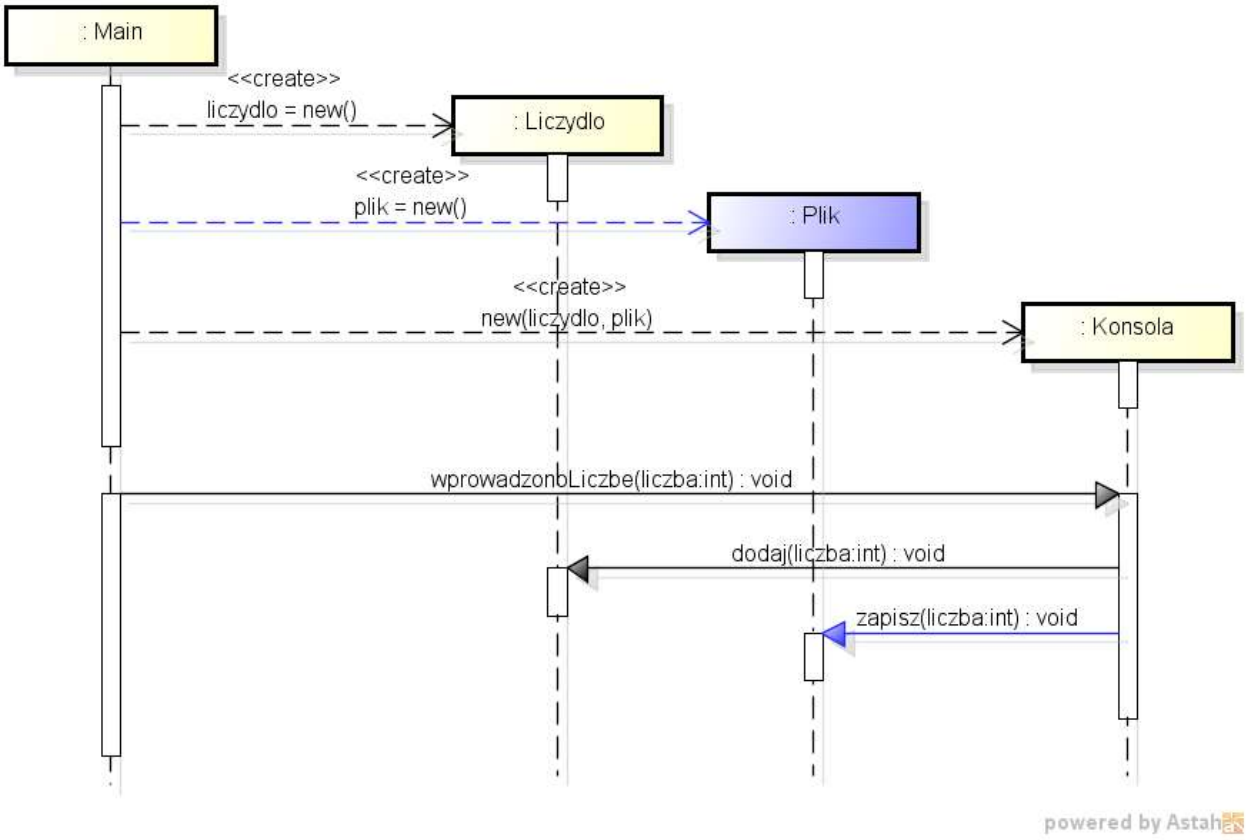
```
class Plik {

   void zapisz(int liczba){

      // zapis do pliku

   }

}
```

Oczywiście należy teraz rozbudować implementację metody wprowadzonoLiczbę() o powiadomienie kolejnego obiektu:

```
class Konsola {
   private Liczydło liczydlo;
   private Plik plik;;

 Konsola(Liczydlo liczydlo, Plik plik){
    this.liczydlo = liczydlo;
    this.plik = plik;
 }



   void wprowadzonoLiczbe(int liczba){

      liczydlo.dodaj(liczba);
      plik.zapisz(liczba);

   }

}


// ...
public static void main(String[] args){
   Liczydlo liczydlo = new Liczydlo();
   Plik plik = new Plik();
   Konsola konsola = new Konsola(liczydlo, plik);

   konsola.wprowadzonoLiczbe(5);
}
// ...
```

Sytuacja ta przedstawiona na diagramach UML wygląda następująco:





Wraz z rozbudową programu mogą pojawić się kolejne obiekty wymagające synchronizacji. Każdy z nich musi zostać dodany do klasy Konsola zwiększając jej stopień skomplikowania. Co więcej obiekty klasy Konsola są na stałe związane z obiektami klasy Liczydło i Plik, co przeczy wzorcowi low coupling.

W rzeczywistych systemach informatycznych liczba obiektów, które wymagają synchronizacji może sięgać tysięcy, a nawet więcej. Przy takiej liczbie obiektów do synchronizacji kod klasy Konsola zostałby zdominowany przez wywołania metod informujących o zmianie stanu. Dlatego dużo lepszym rozwiązaniem takiej sytuacji jest zastosowanie wzorca Obserwator, który pozwala rozwiązać zarówno problem wzrostu złożoności klasy Konsola jak i problem ścisłego powiązania klas.

```
class Konsola {

   private List<Obserwator> obserwatorzy = new ArrayList<>();

   void wprowadzonoLiczbe(int liczba) {

      powiadom(liczba);

   }
   public void obserwuj(Obserwator o) {
      obserwatorzy.add(o);
   }
   private void powiadom(int liczba) {
      for(Obserwator o : obserwatorzy) {
         o.nowaLiczba(liczba);
      }
   }

}

interface Obserwator {
   void nowaLiczba(int liczba);
}
```

Jak widać klasa Konsola zmieniła się znacznie. Dodano dwie nowe metody:

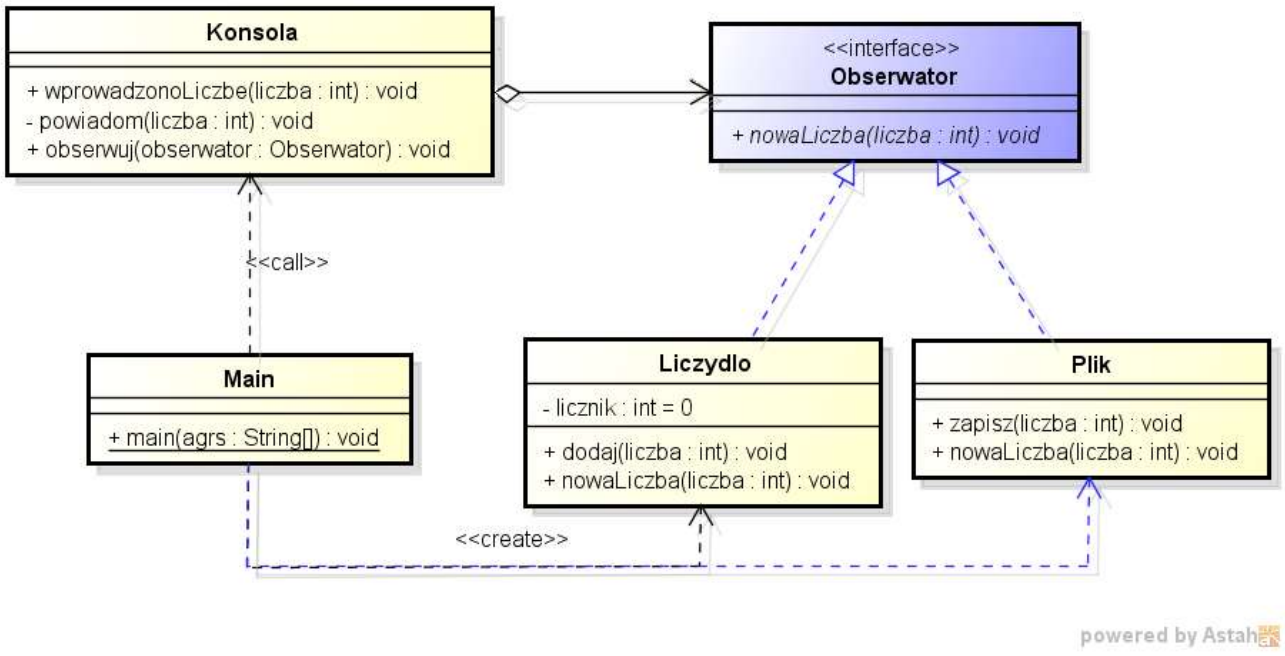
- obserwuj() - służącą innym obiektom do zgłaszania chęci synchronizacji z obiektem klasy Konsola,
- powiadom() - służącą obiektowi klasy Konsola to informowania wszystkich zainteresowanych o zmianie stanu.

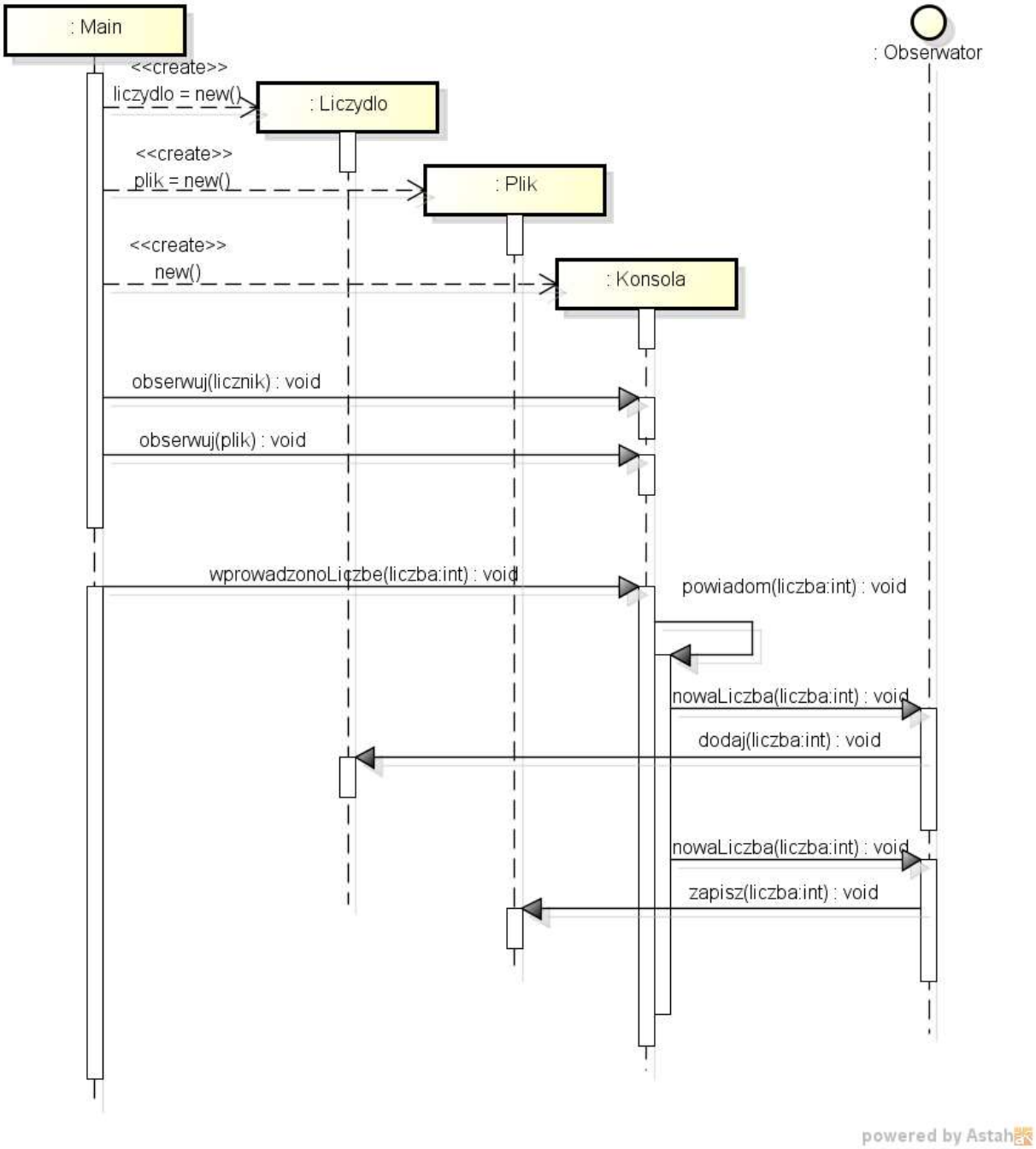
W obecnej postaci nie ma potrzeby modyfikowania jej kodu aby dodać nowy obiekt do synchronizacji stanu. Co więcej klasa Konsola nie jest powiązania przez bezpośrednią referencje zarówno z klasą Liczydlo jak i Plik.

Metoda główna programu inicjalizuje wszystkie niezbędne obiekty (linie 1-3). Klasy Liczydlo i Plik implementują interfejs Obserwatora dzięki czemu mogą otrzymywać powiadomienia o zmianach stanu Konsoli.

```
class Liczydlo implements Obserwator {

   private int licznik = 0;

   public void dodaj(int liczba){

      licznik += liczba;

   }

   public void nowaLiczba(int liczba) {
      dodaj(liczba);
   }

}

class Plik implements Obserwator {

   void zapisz(int liczba){

      // zapis do pliku

   }

   public void nowaLiczba(int liczba) {
      zapis(liczba);
   }

}

// ...

public static void main(String[] args){
   Liczydlo liczydlo = new Liczydlo();
   Plik plik = new Plik();
   Konsola konsola = new Konsola();

   konsola.obserwuj(liczydlo);

   konsola.obserwuj(plik);

   konsola.wprowadzonoLiczbe(5);
}
// ...
```

W języku UML implementacja wzorca obserwator wygląda następująco:





Wraz z rozwojem systemu może pojawić się nowe źródło liczb. Przykładowo do systemu dodano klasę GeneratorLiczb:

```
class GeneratorLiczb {

   private List<ObserwatorGeneratora> obserwatorzy = new ArrayList<>();
   private Random random = new Random();

   void wygenerujLiczbe() {
      int liczba = random.nextInt();
      powiadom(liczba);

   }
   public void obserwuj(ObserwatorGeneratora o) {
      obserwatorzy.add(o);
   }
   private void powiadom(int liczba) {
      for(ObserwatorGeneratora o : obserwatorzy) {
         o.wygenerowano(liczba);
      }
   }

}

interface ObserwatorGeneratora {
   void wygenerowano(int liczba);
}
```

Zastosowanie wzorca obserwator w takiej sytuacji wymaga aby zarówno Liczydlo jak i Plik obserwowały oba źródła liczb.

```
class Liczydlo implements Obserwator, ObserwatorGeneratora {

   private int licznik = 0;

   public void dodaj(int liczba){

      licznik += liczba;
```

```
   }

   public void nowaLiczba(int liczba) {
      dodaj(liczba);
   }
   public void wygenerowano(int liczba){
      dodaj(liczba);
   }

}

class Plik implements Obserwator, ObserwatorGeneratora{

   void zapisz(int liczba){

      // zapis do pliku

   }

   public void nowaLiczba(int liczba) {
      zapis(liczba);
   }

  public void wygenerowano(int liczba){
     dodaj(liczba);
  }



}

// ...

public static void main(String[] args){
   Liczydlo liczydlo = new Liczydlo();
   Plik plik = new Plik();
   Konsola konsola = new Konsola();
   GeneratorLiczb generator = new GeneratorLiczb();

   konsola.obserwuj(liczydlo);

   konsola.obserwuj(plik);

   generator.obserwuj(liczydlo);
   generator.obserwuj(plik);

   konsola.wprowadzonoLiczbe(5);
   generator.wygenerujLiczbe();
}
// ...
```

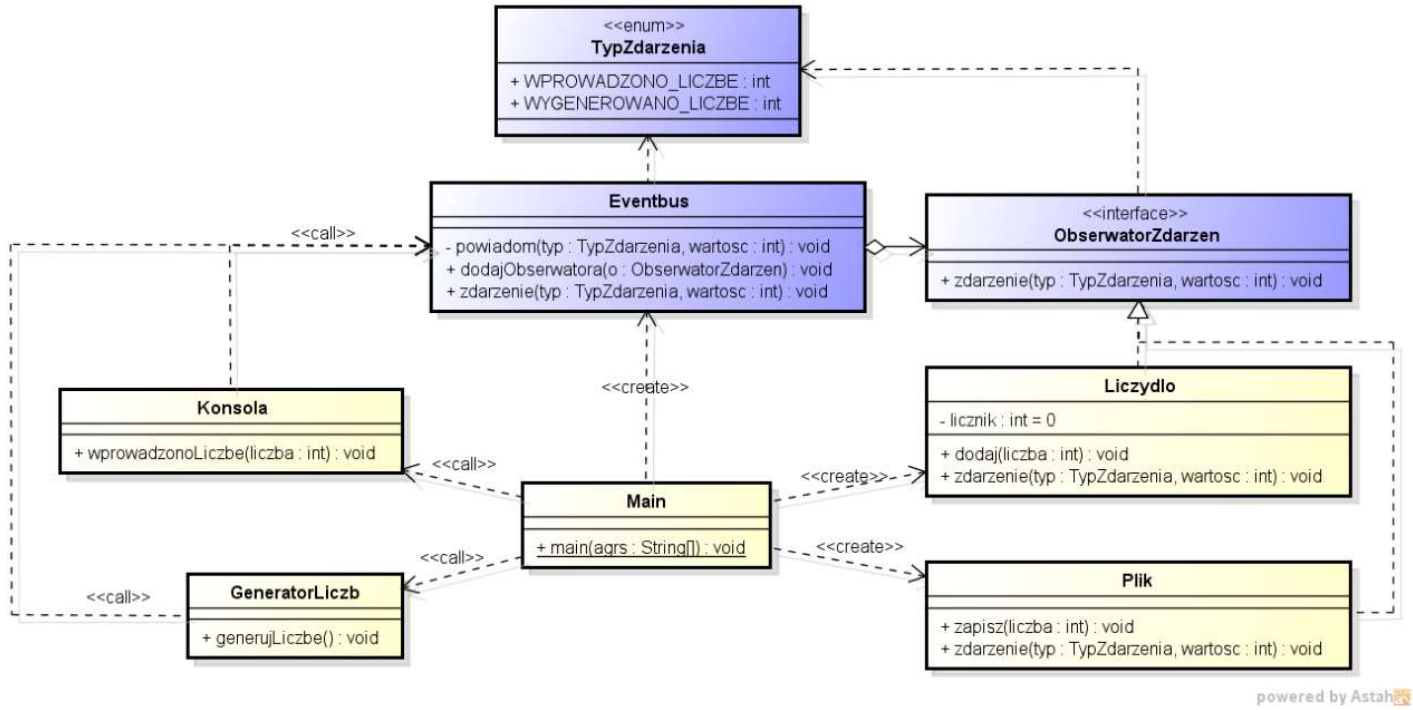
Jak widać dla każdego źródła zmian konieczne jest dodanie nowego obserwatora. Oczywiście ten przykład można by zrealizować znacznie prościej, tak aby nie było konieczne dodawanie nowego interfejsu ObserwatorGeneratora. Jednak w wielu przypadkach takie uproszczenie nie jest możliwe. W przypadku, gdy w systemie mamy wiele obiektów, które chcą nasłuchiwać na zmiany pochodzące z wielu źródeł dobrze jest zastosować inny wzorzec projektowy.

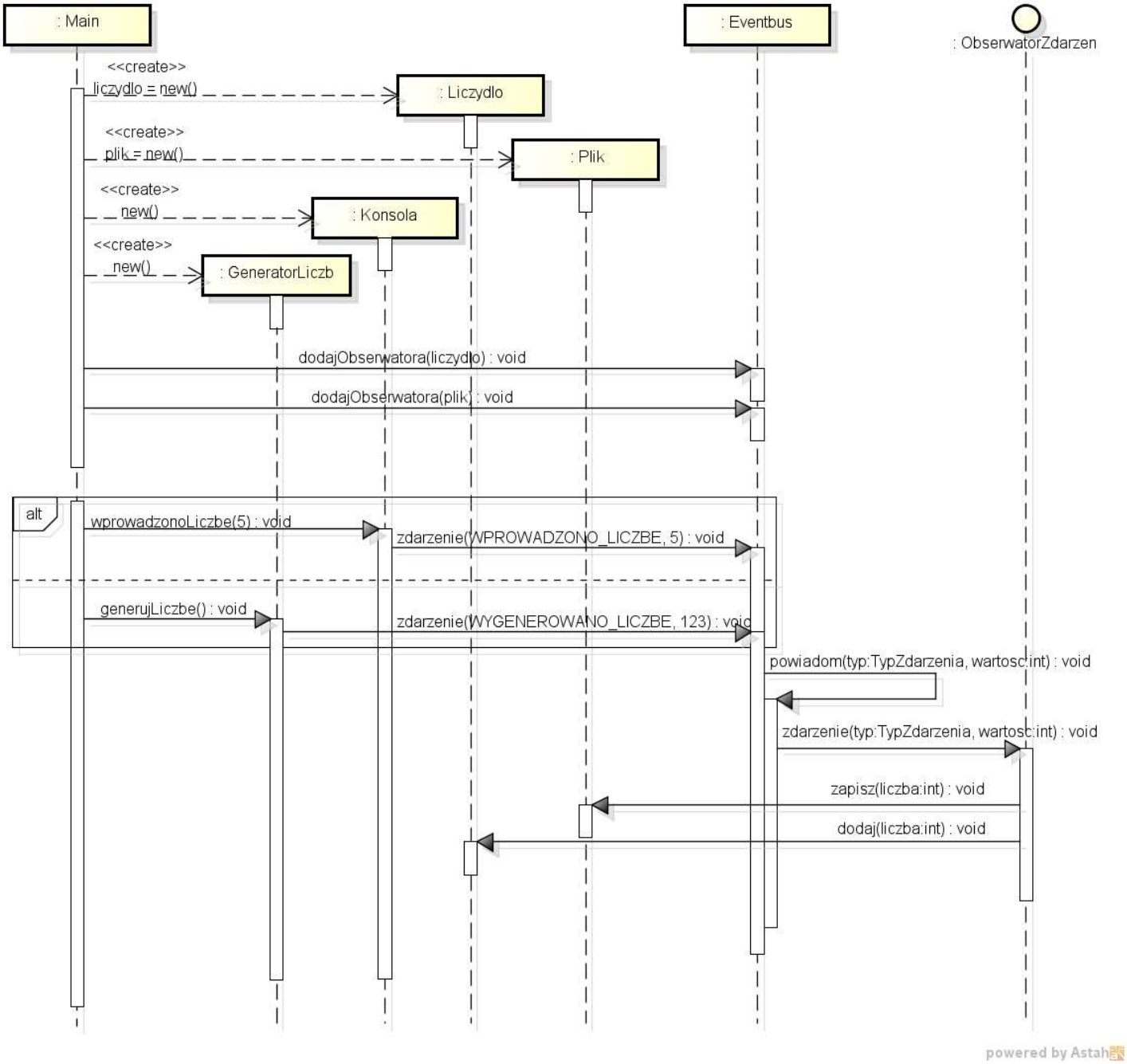
Wzorzec szyny zdarzeń (ang. Eventbus) jest uogólnioną wersją wzorca obserwator. W podstawowej wersji wzorca obserwator każde źródło zmian zarządza osobną pulą obserwatorów, a każdy obiekt zainteresowany otrzymaniem informacji o zmianie stanu rejestruje się we wszystkich dostępnych źródłach. Szyna zdarzeń do nowy obiekt, którego jedynym zadaniem jest przekazywanie informacji o zmianie stanu pomiędzy wieloma źródłami i wieloma odbiorcami. Źródło zmiany stanu, zgłasza taki fakt szynie zdarzeń, która z kolei przekazuje taką informację wszystkim zainteresowanym obiektom. Szyna może obsługiwać różne zdarzenia pochodzące od różnych obiektów, stąd potrzeba rozróżnienia zdarzeń. Najczęściej wprowadza się specjalny typ wyliczeniowy, który określa wszystkie możliwe rodzaje zdarzeń. W naszym przykładzie możliwe są dwa zdarzenia: WPROWADZONO_LICZBE i WYGENEROWANO_LICZBE. Obserwatorzy (obiekty zainteresowane zmianą stanu) zostają powiadomieni o wszystkich zdarzeniach, a następnie na podstawie ich typu mogą określić swoje zachowanie.

Implementacja wzorca szyny zdarzeń w naszym przykładzie oraz odpowiadające jej diagramy UML wyglądają następująco:

```
class Eventbus {
   private static Eventbus domyslny = new Eventbus();

   private List<ObserwatorZdarzen> obserwatorzy = new ArrayList<>();

   public static Eventbus domyslny() {
      return domyslny;
   }

   public void zdarzenie(TypZdarzenia typ, int wartosc) {
      powiadom(typ,wartosc);
   }
   public void dodajObserwatora(ObserwatorZdarzen o) {
      obserwatorzy.add(o);
   }
   private void powiadom(TypZdarzenia typ, int wartosc) {
      for(ObserwatorZdarzen o : obserwatorzy) {
         o.zdarzenie(typ, wartosc);
      }
   }
}

interface ObserwatorZdarzen {
   void zdarzenie(TypZdarzenia typ, int wartosc);
}
enum TypZdarzenia {
   WPROWADZONO_LICZBE,
```

```
    WYGENEROWANO_LICZBE;
}



class Konsola {

   void wprowadzonoLiczbe(int liczba) {

      Eventbus.domyslny().zdarzenie(TypZdarzenia.WPROWADZONO_LICZBE, liczba);
   }
}
class GeneratorLiczb {
   private Random random = new Random();

   void wygenerujLiczbe() {
      int liczba = random.nextInt();
      Eventbus.domyslny().zdarzenie(TypZdarzenia.WYGENEROWANO_LICZBE, liczba);

   }
}

class Liczydlo implements ObserwatorZdarzen {

   private int licznik = 0;

   public void dodaj(int liczba){
      licznik += liczba;
   }

   public void zdarzenie(TypZdarzenia typ, int wartosc) {
      if(typ == TypZdarzenia.WPROWADZONO_LICZBE || typ == TypZdarzenia.WYGENEROWANO_LICZBE) {
         dodaj(liczba);
      }
   }

}

class Plik implements ObserwatorZdarzen {

   void zapisz(int liczba){

      // zapis do pliku

   }

   public void nowaLiczba(int liczba) {
      if(typ == TypZdarzenia.WPROWADZONO_LICZBE || typ == TypZdarzenia.WYGENEROWANO_LICZBE) {
         zapis(liczba);
      }
   }

}


// ...

public static void main(String[] args){
   Liczydlo liczydlo = new Liczydlo();
   Plik plik = new Plik();
   Konsola konsola = new Konsola();
   GeneratorLiczb generator = new GeneratorLiczb();

   Eventbus.domyslny().dodajObserwatora(liczydlo);
   Eventbus.domyslny().dodajObserwatora(plik);

   konsola.wprowadzonoLiczbe(5);
   generator.wygenerujLiczbe();
}
// ...
```


<<enum>>
TypZdarzenia
+ WPROWADZONO_LICZBE : int
+ WYGENEROWANO_LICZBE : int
Eventbus
- powiadom(typ : TypZdarzenia, wartosc : int) : void
+ dodajObserwatora(o : ObserwatorZdarzen) : void
+ zdarzenie(typ : TypZdarzenia, wartosc : int) : void
<<interface>>
ObserwatorZdarzen
+ zdarzenie(typ : TypZdarzenia, wartosc : int) : void
<<call>>
<<create>>
Konsola
+ wprowadzonoLiczbe(liczba : int) : void
Liczydlo
- licznik : int = 0
+ dodaj(liczba : int) : void
+ zdarzenie(typ : TypZdarzenia, wartosc : int) : void
Main
+ main(agrs : String[]) : void
GeneratorLiczb
+ generujLiczbe() : void
Plik
+ zapisz(liczba : int) : void
+ zdarzenie(typ : TypZdarzenia, wartosc : int) : void
powered by Astah


: Main
: Eventbus
: ObserwatorZdarzen
<<create>>
liczydlo = new()
: Liczydlo
<<create>>
plik = new()
: Plik
<<create>>
new()
: Konsola
<<create>>
new()
: GeneratorLiczb
dodajObserwatora(liczydlo) : void
dodajObserwatora(plik) : void
alt
wprowadzonoLiczbe(5) : void
zdarzenie(WPROWADZONO_LICZBE, 5) : void
generujLiczbe() : void
zdarzenie(WYGENEROWANO_LICZBE, 123) : void
powiadom(typ:TypZdarzenia, wartosc:int) : void
zdarzenie(typ:TypZdarzenia, wartosc:int) : void
zapisz(liczba:int) : void
dodaj(liczba:int) : void
powered by Astah

Wzorzec szyny zdarzeń nie tylko rozdzielił klasy będące źródłem zdarzeń (jak we wzrocu obserwator) ale doprowadził do sytuacji, w której obserwatorzy nie muszą mieć nawet dostępu do źródła zdarzeń. Co więcej możliwe jest nasłuchiwanie na zdarzenia pochodzące z klas, które nie są nawet dostępne na etapie kompilacji. Taka sytuacja jest bardzo pożądana z punktu widzenia wzorca low coupling.