**IO**

**Model**:

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

W naukach ścisłych poprzez model rozumiemy pewien zbiór ogólnych założeń, pojęć i zależności pozwalający w uproszczony sposób opisać wybrany aspekt rzeczywistości. Modelem jest również reprezentacja otaczającego świata w umyśle człowieka, której jednak nie należy mylić z rzeczywistością.

**Strzałki:**

* **Dziedziczenie (generalizacja):**



Strzałka powinna być **skierowana w stronę klasy bazowej ( klasy pojęciowo niezależnej )**

* **Zależność:**



w kierunku klasy niezaleznej, współpraca odbywa się na poziomie **metod**

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

Ogólnie powiemy, iż z zależnością klas mamy do czynienia wówczas, gdy:

• na liście argumentów funkcji składowej klasy występuje nazwa innej klasy,

• na liście argumentów funkcji składowej klasy występuje wskaźnik do innej klasy,

• na liście argumentów funkcji składowej klasy występuje odwołanie (referencja) do innej klasy,

• w ciele funkcji składowej klasy występuje zmienna, typu innej klasy,

• w ciele funkcji składowej klasy występuje wskaźnik do innej klasy,

• w ciele funkcji składowej klasy występuje odwołanie (referencja) do innej klasy,

• typem powrotnym funkcji składowej klasy jest wskaźnik do innej klasy

<<cal>> – operacje w klasie C wywołują operacje z klasy A,

<<create>> – klasa A tworzy egzemplarz klasy B,

<<instantiate>> – obiekt klasy D jest egzemplarzem (instancją) klasy B,

<<use>> – do zaimplementowania klasy C wymagana jest klasa D.

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Автоматически созданное описание

* **Powiązanie:**



Linia wskazuje na kierunek powiązania.

Jeżeli w przypadku każdej pary klas elementy jednej z nich mogą modyfikować zachowanie elementów drugiej i na odwrót, mamy do czynienia z powiązaniem !

Powiązanie jest związkiem strukturalnym wskazującym, że obiekty jednego typu są połączone z obiektami innego typu **poprzez atrybuty danych klas**. Na poziomie implementacji określa się tak głównie sytuacje, w których jedna z klas zawiera w swoich polach referencję/wskaźnik na obiekt innej klasy.

W implementacji C++ powiązanie realizuje się poprzez deklarację na liście atrybutów jednej klasy zmiennej typu lub wskaźnika do innej klasy.

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

Wyobraź sobie, że z klocków LEGO budujesz dom i samochód. Dom i samochód to jak przyjaciele — jeden zna drugiego, ale mogą też być osobno. Nawet jeśli samochód pojedzie daleko, dom wciąż stoi na swoim miejscu.

Powiązanie dwustronne:



* **Agregacja (**agregacja prosta**):**



Romb jest po stronie **całości**.

Jest szczególnym przypadkiem **powiązania**, związek „ całość – część ”. Agregacja umożliwia oddzielenie całości od części. Obiekty obu klas mogą istnieć niezależnie od siebie. W praktyce zwykle wiąże się to z sytuacją, w której jedna z klas zawiera modyfikowalną kolekcję obiektów innej klasy.

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описаниеИзображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

Teraz pomyśl, że w twoim domku z klocków mieszkają ludziki LEGO. Ludziki są częścią domu, ale jeśli zechcesz, możesz wziąć ludzika i zabawić się nim osobno, bez domu. Ludziki i dom są jak rodzina, która dobrze się razem bawi, ale czasami członkowie rodziny robią też rzeczy samodzielnie.

* **Kompozycja (** agregacja całkowita **):**



Romb jest po stronie **całości**.

Tutaj mówimy o sytuacji, w której jedna z klas stanowi **nieodłączną** część składową innej.

Proces konstrukcji całości musi być poprzedzony skonstruowaniem wszystkich jej elementów składowych (części nie mogą powstawać po utworzeniu całości, ale w momencie gdy destrukcji podlega całość muszą zginąć wszystkie jej części). W przypadku **agregacji prostej** część może zależeć od **kilku całości,** zaś w przypadku **kompozycji** tylko od **jednej**.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

książka o podanej sygnaturze może należeć tylko do **konkretnej** biblioteki.

! Klasa TLibrary (biblioteka) całkowicie agreguje klasę TBook dzięki odpowiedniemu zaimplementowaniu konstruktora TLibrary(...) i destruktora ~TLibrary(). Obiekt klasy TBook tworzony jest w **konstruktorze** klasy TLibrary oraz niszczony jest w momencie wywołania jej destruktora.

**Liczebność powiązania:**

Podając liczebność przy jednym końcu powiązania wskazujemy ile obiektów jednej klasy powinno być połączonych z każdym obiektem klasy znajdującej się na drugim końcu powiązania. Liczebność można ustalić poprzez wyszczególnienie żądanej liczby, np.: 5, przedziału liczbowego 1..5 (od 1 do 5), dowolnie wiele (0..\*) albo co najmniej dwa (2..\*).

**Diagramy**:

1. **Diagram czynności (aktywności):**

**Изображение выглядит как диаграмма, текст, линия, дизайн

Автоматически созданное описание**Po pierwsze, wykorzystujemy je do modelowania przepływu czynności między poszczególnymi częściami systemu. W tego rodzaju przypadkach diagram czynności można skojarzyć z klasą, komponentem, interfejsem, przypadkiem użycia itp. Po drugie, diagramy te mogą być pomocne do modelowania operacji. Na tego rodzaju diagramach pojęcie operacji może być dwojako rozumiane. Modelowanym bytem może być pojedyncza funkcja lub zespół funkcji i/lub instrukcji wykonywanych np. w głównej funkcji main(), gdzie przedstawiać można szczegóły przeprowadzanych obliczeń.

1. **Diagram obiektów:**

**Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание**

Jeżeli np. klasa TStudent jest powiązana z klasą Tbook, to między ich obiektami może wystąpić **wiązanie**. Na ogół wiązania (między obiektami) są egzemplarzami powiązań (między klasami).**Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание**

Изображение выглядит как текст, Шрифт, Прямоугольник, линия

Автоматически созданное описание

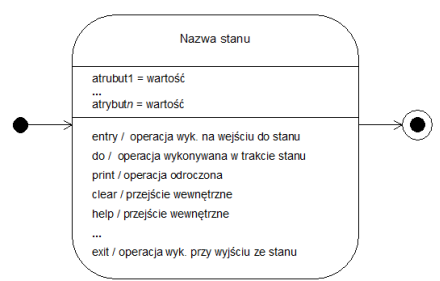
Изображение выглядит как текст, Шрифт, белый, линия

Автоматически созданное описание

Standardowe **stereotypy wiązań** to:

* <<association>>– umieszczony na jednym końcu wiązania wskazuje, że odpowiadający obiekt jest widoczny przez to wiązanie,
* <<global>>– umieszczony przy obiekcie na jednym końcu wiązania oznacza, że obiekt ten jest widoczny, ponieważ jest w otaczającym zasięgu,
* <<local>>– umieszczony przy obiekcie na jednym końcu wiązania oznacza, że obiekt ten jest widoczny, ponieważ jest w lokalnym zasięgu,
* <<parametr>>– umieszczony przy obiekcie na jednym końcu wiązania oznacza, że obiekt ten jest widoczny, ponieważ jest parametrem,
* <<self>>– umieszczony przy obiekcie na jednym końcu wiązania oznacza, że obiekt ten jest widoczny, ponieważ to właśnie on odebrał zlecenie wykonania danej operacji.

1. **Diagram stanów:**

****Symbol stanu oznaczany jest prostokątem z zaokrąglonymi narożnikami. Czarne kółko oznacza punkt wejścia do stanu, strzałka jest symbolem przejścia, zaś kółko z czarną kropką (tzw. bycze oko) oznacza punkt wyjścia ze stanu.

Diagramy stanów obrazują sposób, w jaki elementy modelu zmieniają w czasie swoje własności w odpowiedzi na różnego rodzaju zdarzenia i interakcje.

**Изображение выглядит как текст, диаграмма, снимок экрана, линия

Автоматически созданное описание**

1. Изображение выглядит как текст, снимок экрана, Шрифт, линия

   Автоматически созданное описаниеИзображение выглядит как текст, диаграмма, Параллельный, План

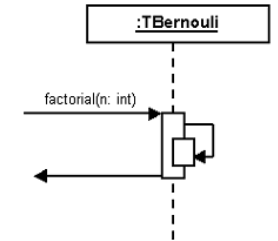
   Автоматически созданное описание**Diagram sekwencji ( przebiegu ):**

* 1) synch. lub blokująca -- która powoduje oczekiwanie obiektu wywołującego na jej zakończenie (aby móc realizować kolejne instrukcje)
* 3) asynch. lub nieblokująca -- która zwraca sterowanie natychmiast po jej wywołaniu
* **<<create>> i <<destroy>>** -- komunikaty **asynchroniczne!**
* **<<destroy>> -- wywolanie *delete***
* **ob2 i ob1 – obiekty osierocone** ( na diag obiektów )
* **ob2 – obiekt aktywny** ( na diag obiektów )

**Изображение выглядит как текст, диаграмма, Параллельный, План

Автоматически созданное описание**

**🡨 diagram do kodu z 1 prez (1.7)**

**🡨 wywołanie funkcji rekurencyjnej**

**Bloki i operatory interakcji:**

Na diagramach sekwencji taką grupę operacji obejmuje się prostokątem, w którego lewym górnym narożniku, w pięciokącie umieszcza się słowo kluczowe lub opis określający znaczenie danego bloku (tzw. operator interakcji).

**Изображение выглядит как диаграмма, План, Технический чертеж, Прямоугольник

Автоматически созданное описание🡨 oznaczenie pętli *while* i warunku *if***

**Operatory interakcji:**

• **alt** (alternative) - odpowiadający instrukcji if-else;

• **opt** (optional) - reprezentujący instrukcję if (bez else),

• par (parallel) - nakazujący wykonać operacje równolegle,

• critical - blok atomowy, oznaczający obszar krytyczny o najwyższym priorytecie wykonania. Obiekty nie mogą uczestniczyć w innych zadaniach.

• **loop** - definiujący pętlę typu for lub while

• **break** - wykonanie fragmentu i zakończenie interakcji,

• neg – funkcjonalności nieprawidłowe – wskazuje na wyjątki, które muszą zostać obsłużone,

• strict – ścisłe uporządkowanie komunikatów,

• seq - słabe uporządkowanie dotyczy komunikatów z kilku lini mogących wystąpić w dowolnym porządku,

• ignore – komunikaty nie mają istotnego wpływu na całość komunikacji,

• consider – istotność – komunikaty muszą zostać wykonane.

1. **Diagram komunikacji (kooperacji) :**

**Изображение выглядит как текст, линия, Шрифт, снимок экрана

Автоматически созданное описание🡨 do wyzej diagramu sekwencji**

Umożliwia pokazanie kolejności wysyłania komunikatów

**Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание**

**🡨 wysyłanie komunikaku do wielu obiektów tej samej klasy**

Warunek pętli *while* oznaczamy gwiazdką

1. **Изображение выглядит как текст, линия, диаграмма, снимок экрана

   Автоматически созданное описаниеИзображение выглядит как текст, круг, снимок экрана, Шрифт

   Автоматически созданное описаниеDiagram współpracy:**
2. **Diagram harmonogramowania zadań:**

**Изображение выглядит как снимок экрана, текст, диаграмма, линия

Автоматически созданное описание**

**Metodyka**:

**Co to?**

Ustandaryzowany zbiór procedur i procesów stosowany podczas rozwiązywania problemów wynikłych w trakcie projektowania i wdrażania oprogramowania traktowanego jako nieodłączna część określonego systemu informatycznego.

**SDM (**Software Development Methodology**) –** metodyka rozwoju oprogramowania

Albo

**1 --** **SDLC** (System Development Life Cycle) -- cykl rozwoju systemów informatycznych**:**

**Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Автоматически созданное описание**

**-- Metoda biznesowa (proces biznesowy)**

**2 -- Proces (model) kaskadowy ( „waterfall model” ):**

Ścisła interpretacja modelu kaskadowego traktuje poszczególne fazy jako niezależne etapy**Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание**

**Nie ma powrotu do poprzednich faz!**

**3 – Proces (model) iteracyjny:**

**Изображение выглядит как текст, диаграмма, снимок экрана, линия

Автоматически созданное описаниеwiele razy możemy przechodzić przez każdy z tych etapów**

**4 – Model V:**

**Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание**

**Każdy etap projektowania kończy się stworzeniem danych wejściowych dla następnej fazy oraz odpowiadającej fazy testowania.**

**5 – Realizacja iteracyjna i przyrostowa:**

**Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описаниеprojekt jest rozwijany poprzez cykliczne powtarzanie faz projektowych.**

**Kazda iteracja kończy się dostarczeniem działającej części systemu, zwaną przyrostem.**

**Zakłada dostarczenie funkcjonalnych fragmentów produktu (przyrostów) po każdej iteracji.**

**6 – Programowanie odkrywcze:**

**Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание**

**7 – Prototypowanie:**

**Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание**

**Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание8 – GRAPPLE:**

**Powstał jako próba uproszczenia SDLC,**

**Utożsamia się z RAD (Rapid Application Development)**

**9 – XP – Programowanie ekstremalne:**

**Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описаниеnależy do rodziny metodyk zwinnych ( Agile )**

**10 – TDD ( Test-Driven Development):**

**Изображение выглядит как текст, круг, дизайн

Автоматически созданное описание**

**1 -- Najpierw test jednostkowy**

**2 – kod aby poprawnie przechodził test**

**3 -- refaktoryzacja**

**11 – Agile :**

**-- a : SCRUM**

**Изображение выглядит как диаграмма, зарисовка, линия, План

Автоматически созданное описание**

**Wzorce projektowe**: rodzaje od najnizszego:

* Programowania lub idiomy
* Projektowe programowania obiektowego
* Architektoniczne
* Analityczne

**Wzorce Projektowe programowania obiektowego**:

* Konstrukcyjne ( wykorzystywane do pozyskiwania obiektów zamiast tworzenia egzemplarzy klas ) :
  + **Budowniczy** ( *builder* ) -- Umożliwia tworzenie złożonych obiektów krok po kroku, izolując konstrukcję obiektu od jego reprezentacji.
  + Metoda wytworcza ( *factory method* ) ?
  + **Prototyp** ( *prototype* ) -- Umożliwia kopiowanie istniejących obiektów bez konieczności zrozumienia szczegółów ich implementacji.
  + **Singleton** ( *singleton* ) -- Zapewnia, że klasa ma tylko jedną instancję i dostarcza globalny punkt dostępu do tej instancji.
  + **Fabryka abstrakcyjna** ( *abstract factory* ) -- Umożliwia tworzenie rodzin powiązanych lub zależnych obiektów bez określania ich konkretnych klas.
* Strukturalne ( pomagające łączyc obiekty w wieksze struktury ) :
  + **Adapter** ( *adapter* ) -- Pozwala obiektom o niekompatybilnych interfejsach współpracować ze sobą, przekształcając interfejs jednego obiektu na interfejs oczekiwany przez drugi.
  + **Dekorator** ( *decorator* ) -- Umożliwia dodawanie nowych funkcji do obiektów dynamicznie, bez zmiany ich interfejsu.
  + Fasada ( *facade* )
  + **Kompozyt** ( *composite* ) -- Umożliwia traktowanie pojedynczych obiektów i kompozycji obiektów (grup) w jednolity sposób.
  + Most ( *bridge* ) ?
  + **Pełnomocnik** ( *proxy* ) -- Dostarcza zastępczy lub pośredniczący obiekt, który kontroluje dostęp do innego obiektu, na przykład do zarządzania jego zasobami lub zabezpieczeniami.
  + Pyłek ( *flyweight* )
* Operacyjne (behawioralne) ( słuzace do definiowania komunikacji pomiedzy obiektami oraz kontrolowania przeplywu danych w złozonym programie ) :
  + Łańcuch odpowiedzialnosci ( *chain of responsibility* )
  + Polecenie ( *command* )
  + Interpreter ( *interpreter* )
  + Iterator ( *iterator* )
  + Mediator ( *mediator* )
  + Memento ( *memento* )
  + **Obserwator** ( *observer* ) -- Umożliwia obiektom subskrybowanie i reagowanie na zdarzenia występujące w innym obiekcie.
  + **Stan** ( *state* ) -- Pozwala obiektowi zmieniać swoje zachowanie, gdy zmienia się jego wewnętrzny stan.
  + **Strategia** ( *strategy* ) -- Umożliwia definiowanie rodziny algorytmów, enkapsulację każdego z nich osobno i ich wymianę między sobą.
  + **Metoda szablonu** ( *template method* ) -- Definiuje szkielet algorytmu w metodzie, przekazując odpowiedzialność za implementację niektórych kroków do klas pochodnych.
  + Wizytator ( *visitor* )

**Podobieństwa:**

* [**Budowniczy**](https://refactoring.guru/pl/design-patterns/builder) koncentruje się na konstruowaniu złożonych obiektów krok po kroku. [**Fabryka abstrakcyjna**](https://refactoring.guru/pl/design-patterns/abstract-factory) specjalizuje się w tworzeniu rodzin spokrewnionych ze sobą obiektów. Fabryka abstrakcyjna zwraca produkt natychmiast, zaś Budowniczy pozwala dołączyć dodatkowe etapy konstrukcji zanim będzie można pobrać finalny produkt.
* Możesz zastosować wzorzec [**Budowniczy**](https://refactoring.guru/pl/design-patterns/builder) by tworzyć złożone drzewa [**Kompozytowe**](https://refactoring.guru/pl/design-patterns/composite) dzięki możliwości zaprogramowania ich etapów konstrukcji tak, aby odbywały się rekurencyjnie.
* [**Fabryki abstrakcyjne**](https://refactoring.guru/pl/design-patterns/abstract-factory), [**Budowniczych**](https://refactoring.guru/pl/design-patterns/builder) oraz [**Prototypy**](https://refactoring.guru/pl/design-patterns/prototype) można zaimplementować jako [**Singletony**](https://refactoring.guru/pl/design-patterns/singleton).
* Projekty intensywnie korzystające ze wzorców [**Kompozyt**](https://refactoring.guru/pl/design-patterns/composite) i [**Dekorator**](https://refactoring.guru/pl/design-patterns/decorator) mogą skorzystać również na zastosowaniu [**Prototypu**](https://refactoring.guru/pl/design-patterns/prototype). Zastosowanie tego wzorca pozwala klonować złożone struktury zamiast konstruować je ponownie od zera.
* Można użyć **Prototypu** do skomponowania metod klas **Fabryki abstrakcyjnej**
* [**Adapter**](https://refactoring.guru/pl/design-patterns/adapter) zapewnia zupełnie inny interfejs dostępu do istniejącego obiektu. Z drugiej strony, w przypadku wzorca [**Dekorator**](https://refactoring.guru/pl/design-patterns/decorator) interfejs albo pozostaje taki sam, albo zostaje rozszerzony. Ponadto, Decorator obsługuje rekurencyjną kompozycję, co nie jest możliwe w przypadku użycia Adapter.
* Za pomocą [**Adapter**](https://refactoring.guru/pl/design-patterns/adapter) można uzyskać dostęp do istniejącego obiektu za pośrednictwem innego interfejsu. W przypadku [**Pełnomocnik**](https://refactoring.guru/pl/design-patterns/proxy) interfejs pozostaje taki sam. Za pomocą [**Dekorator**](https://refactoring.guru/pl/design-patterns/decorator) uzyskuje się dostęp do obiektu za pośrednictwem ulepszonego interfejsu.
* [**Stan**](https://refactoring.guru/pl/design-patterns/state), [**Strategia**](https://refactoring.guru/pl/design-patterns/strategy) (i w pewnym stopniu [**Adapter**](https://refactoring.guru/pl/design-patterns/adapter)) mają podobną strukturę. Wszystkie oparte są na kompozycji, co oznacza delegowanie zadań innym obiektom. Jednak każdy z tych wzorców rozwiązuje inne problemy.
* [**Stan**](https://refactoring.guru/pl/design-patterns/state) można uważać za rozszerzenie **[Strategii](https://refactoring.guru/pl/design-patterns/strategy)**. Oba wzorce oparte są o kompozycję: zmieniają zachowanie kontekstu przez delegowanie części zadań obiektom pomocniczym. Strategia czyni te obiekty całkowicie niezależnymi i nieświadomymi siebie nawzajem. Jednakże Stan nie ogranicza zależności pomiędzy konkretnymi stanami i pozwala im zmieniać stan kontekstu według uznania.
* [**Kompozyt**](https://refactoring.guru/pl/design-patterns/composite) i [**Dekorator**](https://refactoring.guru/pl/design-patterns/decorator) mają podobne diagramy struktur ponieważ oba bazują na rekursywnej kompozycji w celu zorganizowania nieokreślonej liczby obiektów.

*Dekorator* przypomina *Kompozyt*, ale posiada tylko jeden element podrzędny. Ponadto, kolejną różnicą jest to, że *Dekorator* przypisuje dodatkowe obowiązki opakowywanemu obiektowi, zaś *Kompozyt* jedynie “sumuje” wyniki otrzymane od elementów podrzędnych.

Wzorce mogą też współpracować: *Dekorator* może służyć rozszerzeniu zachowania określonego obiektu w drzewie *Kompozytowym*

* [**Dekorator**](https://refactoring.guru/pl/design-patterns/decorator) pozwala zmienić otoczkę obiektu, zaś **[Strategia](https://refactoring.guru/pl/design-patterns/strategy)** jej wnętrze.
* [**Dekorator**](https://refactoring.guru/pl/design-patterns/decorator) i **[Pełnomocnik](https://refactoring.guru/pl/design-patterns/proxy)** mają podobne struktury, ale inne cele. Oba wzorce bazują na zasadzie kompozycji — jeden obiekt deleguje część zadań innemu. Pełnomocnik dodatkowo zarządza cyklem życia obiektu udostępniającego jakąś usługę, zaś komponowanie Dekoratorów leży w gestii klienta.
* [**Metoda szablonowa**](https://refactoring.guru/pl/design-patterns/template-method) polega na mechanizmie dziedziczenia: pozwala zmieniać części algorytmu rozszerzając je w podklasach. **[Strategia](https://refactoring.guru/pl/design-patterns/strategy)** bazuje na kompozycji: można zmienić część zachowania obiektu poprzez nadanie mu różnych strategii odpowiadających temu zachowaniu. Metoda szablonowa działa na poziomie klasy, więc jest statyczna. Strategia działa na poziomie obiektu, więc pozwala przełączać zachowania w trakcie działania programu.