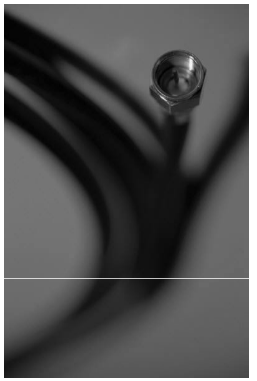
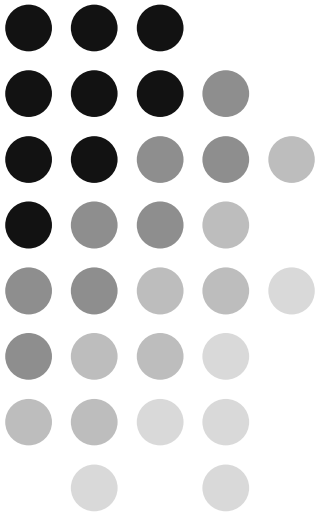
Podstawy Inżynierii Oprogramowania





Modelowanie

Obiektowość w projektowaniu

Powtórne użycie

Modelowanie pojęciowe

- Projektant i programista muszą dokładnie wyobrazić sobie problem oraz metodę jego rozwiązania. Zasadnicze procesy tworzenia oprogramowania zachodzą w ludzkim umyśle i nie są związane z jakimkolwiek językiem programowania.

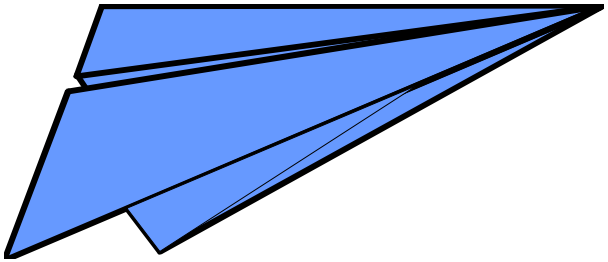
- Pojęcia **modelowania pojęciowego** (*ang. conceptual modeling*) oraz **modelu pojęciowego** (*ang. conceptual model*) odnoszą się procesów myślowych i wyobrażeń towarzyszących pracy nad oprogramowaniem.

- Modelowanie pojęciowe jest wspomagane przez środki wzmacniające ludzką pamięć i wyobraźnię. Służą one do przedstawienia rzeczywistości opisywanej przez dane, procesów zachodzących w rzeczywistości, struktur danych oraz programów składających się na konstrukcję systemu.

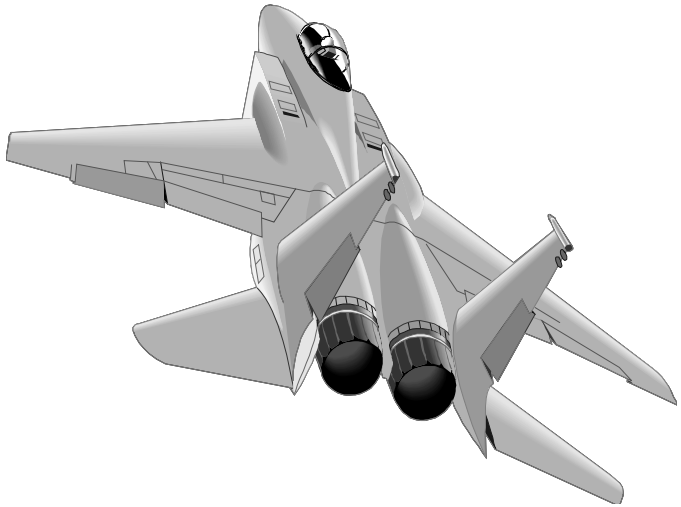
Czy zawsze musimy modelować?

Mniej istotne

Ważniejsze



Papierowy samolot



Myśliwiec

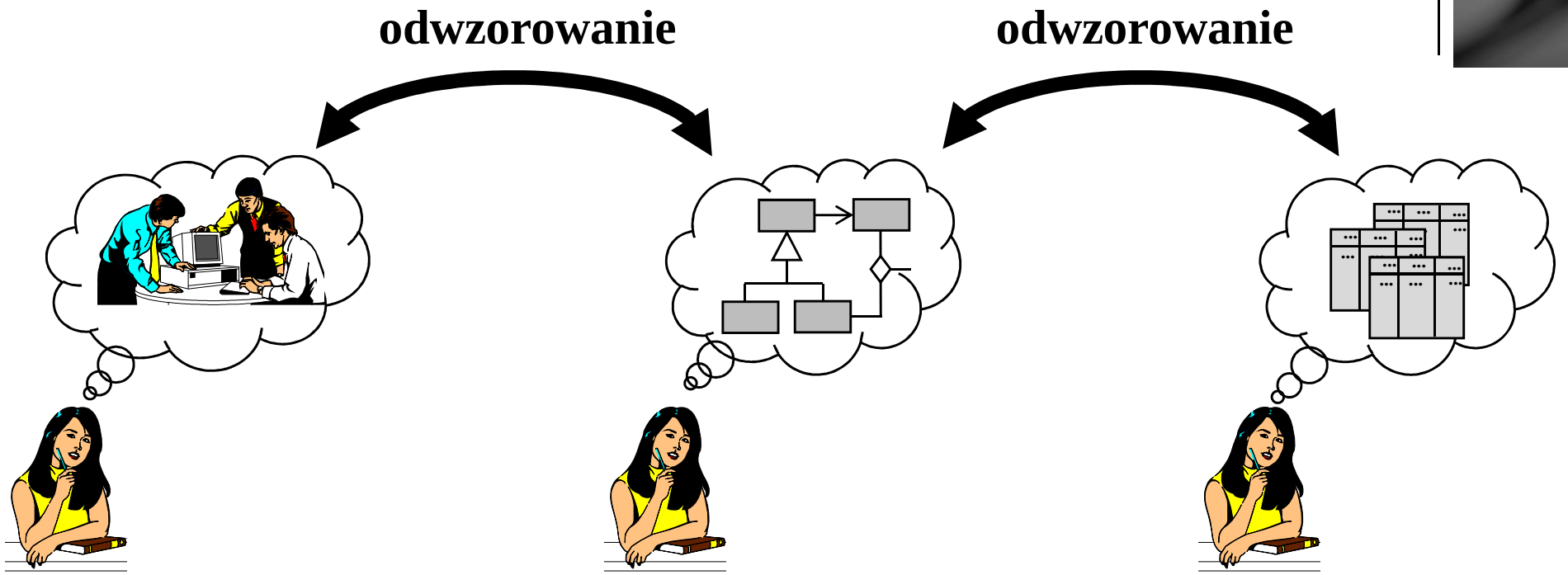
Dlaczego zespoły programistów nie modelują?

- Wiele zespołów podchodzi do wytwarzania złożonego oprogramowania jak do budowy papierowego samolociku.
  - Zaczynają pisać kod na podstawie wymagań.
  - Pracują długie godziny i piszą dłuższy kod programu.
  - Architektura nie jest planowana
    - ... Mówi się, że nad projektami informatycznymi wisi fatum porażki (*ang. doom of failure*).
  - Często jest to związane z traktowaniem myśliwca w kategoriach papierowego samolociku.

Cztery zasady modelowania

- Wybór modelu ma wpływ na to jak będziemy postrzegać rzeczywistość.
- Każdy model może być wyrażony na dowolnym poziomie szczegółowości
- Najlepsze model są bezpośrednio odnoszą się do rzeczywistości
- Pojedynczy model jest niewystarczający.

Perspektywy w modelowaniu pojęciowym



Percepcja rzeczywistego świata

Analityczny model rzeczywistości

Model struktur danych i procesów SI

Trwałą tendencją w rozwoju metod i narzędzi projektowania oraz konstrukcji SI jest dążenie do minimalizacji luki pomiędzy myśleniem o rzeczywistym problemie a myśleniem o danych i procesach zachodzących na danych.

Modelowanie analityczne systemu

- Modelowanie systemu pomaga analitykowi w zrozumieniu funkcjonalności systemu oraz w komunikacji z klientem.
- Modele mogą prezentować system z różnych punktów widzenia
  - **Zewnętrznego** – pokazuje kontekst lub środowisko systemu;
  - **Zachowania** – modeluje zachowanie systemu;
  - **Strukturalnego** – modeluje architekturę systemu lub strukturę przetwarzania danych.

Typy modeli

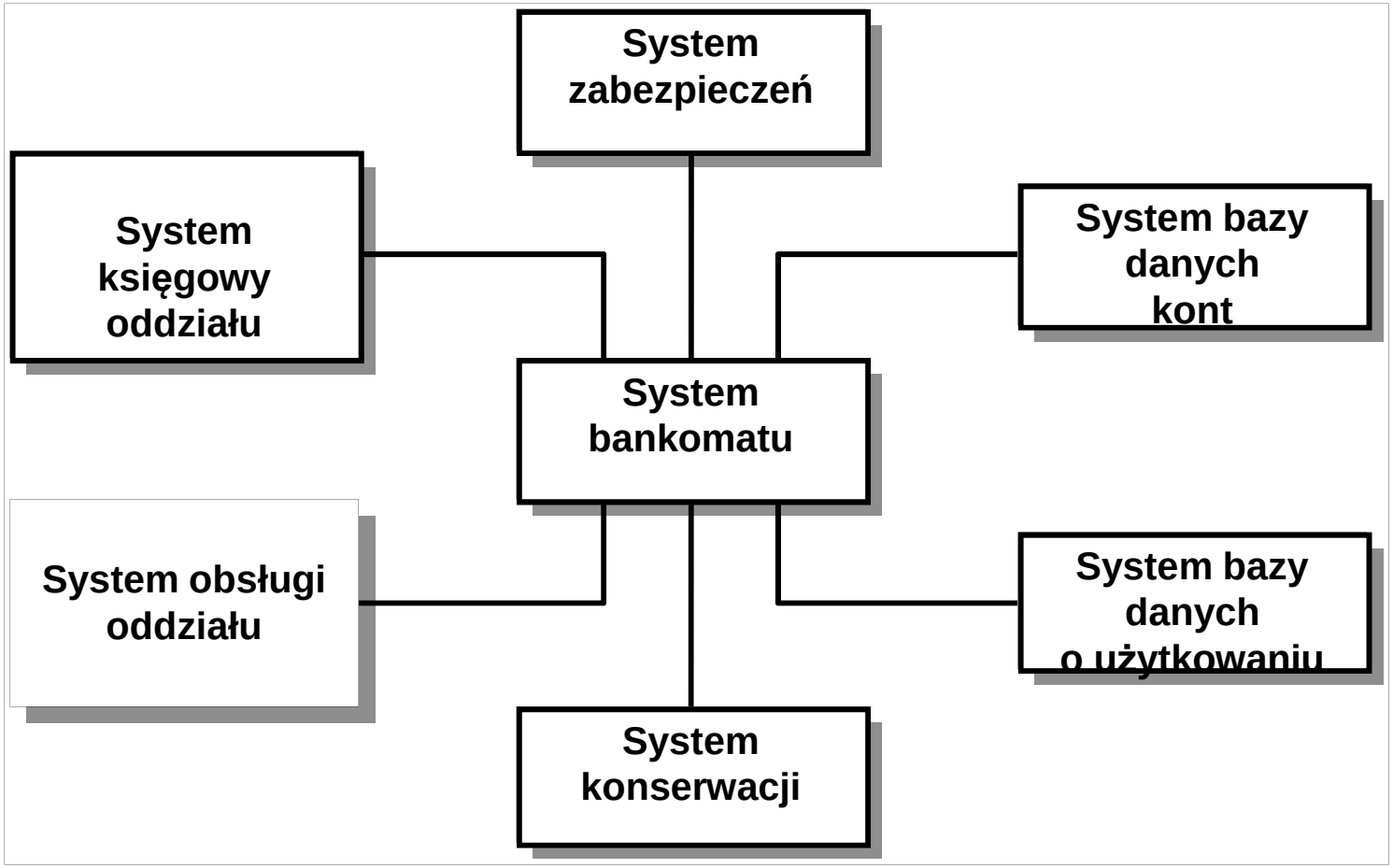
- **Model przetwarzania danych** – jak dane są przetwarzane na różnych etapach pracy systemu.
- **Model kompozycji** –jak elementy systemu komponowane są z mniejszych fragmentów.
- **Model architektoniczny** – z jakich zasadniczych podsystemów składa się system.
- **Model klasyfikacji** – wspólne cechy poszczególnych elementów systemu.
- **Model bodziec-reakcja** – w jaki sposób system reaguje na zdarzenia zachodzące tak poza nim jak i w jego wnętrzu.

Modele kontekstowe

Context models

- Ilustrują operacyjny kontekst systemu-otoczenie.
- Kwestie społeczne i organizacyjne mogą mieć wpływ na ustalenia co do systemu należy a co jest poza jego granicami.
- Modele architektoniczne pokazują system i jego powiązania z innymi systemami.

Kontekst systemu bankomatu



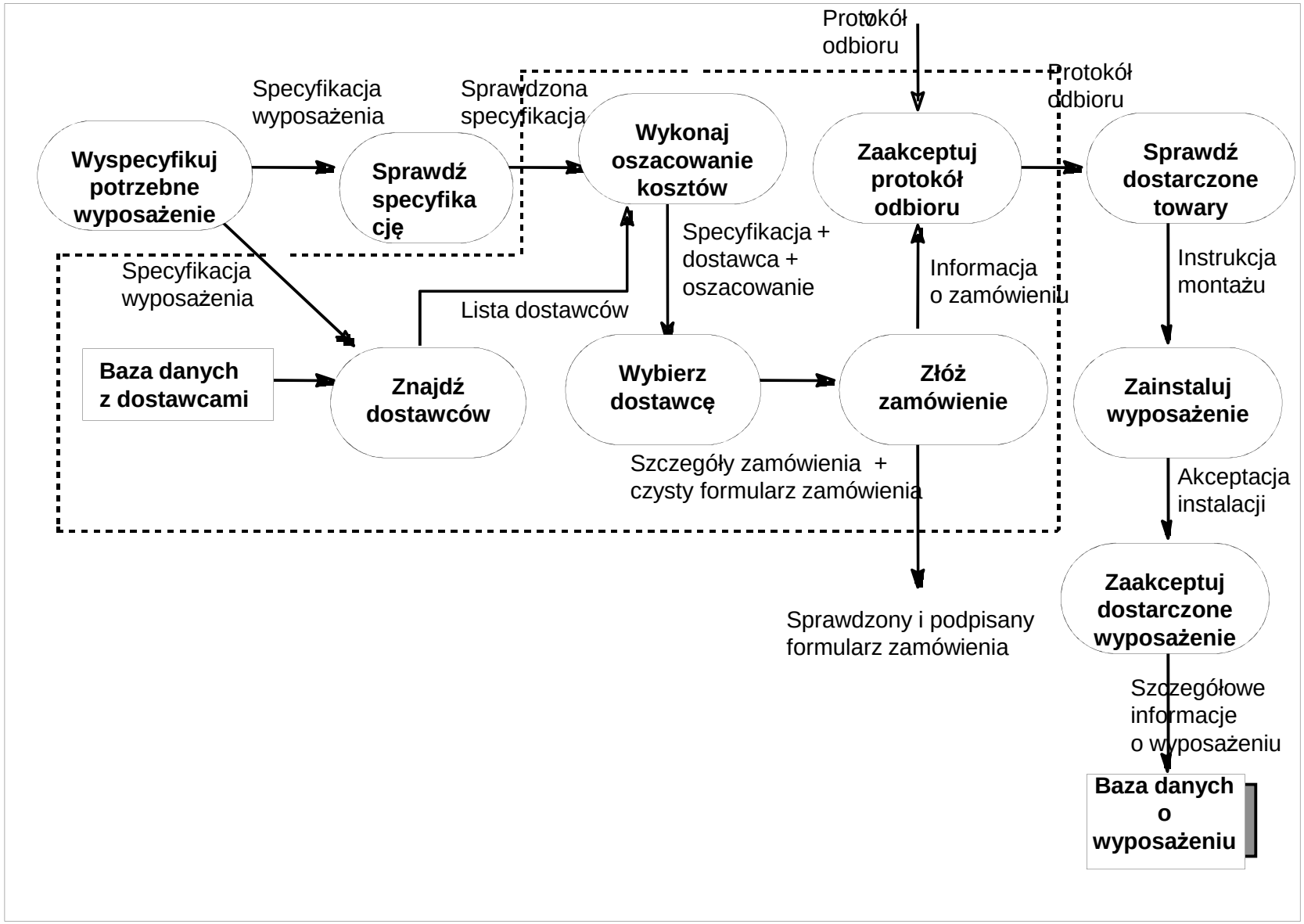
Modele procesu

Process models

- Pokazują czynności wspierane przez system.

- Uzupełniają model kontekstowy i pomagają w podjęciu decyzji, które czynności będą wykonywane automatycznie.

Model procesu zakupu wyposażenia dla firmy



Modele zachowania

- Opisują ogólne zachowanie systemu.
- Typy:
  - **Modele przetwarzania (przepływu) danych** – pokazują sposób w jaki dane są przetwarzane i jak przepływają przez system;
  - **Modele stanów** (maszyna stanów) pokazujące reakcje systemu na zdarzenia.
- Pozwalają spojrzeć na zachowanie systemu z różnych punktów widzenia.

Maszyny stanów

State machine models

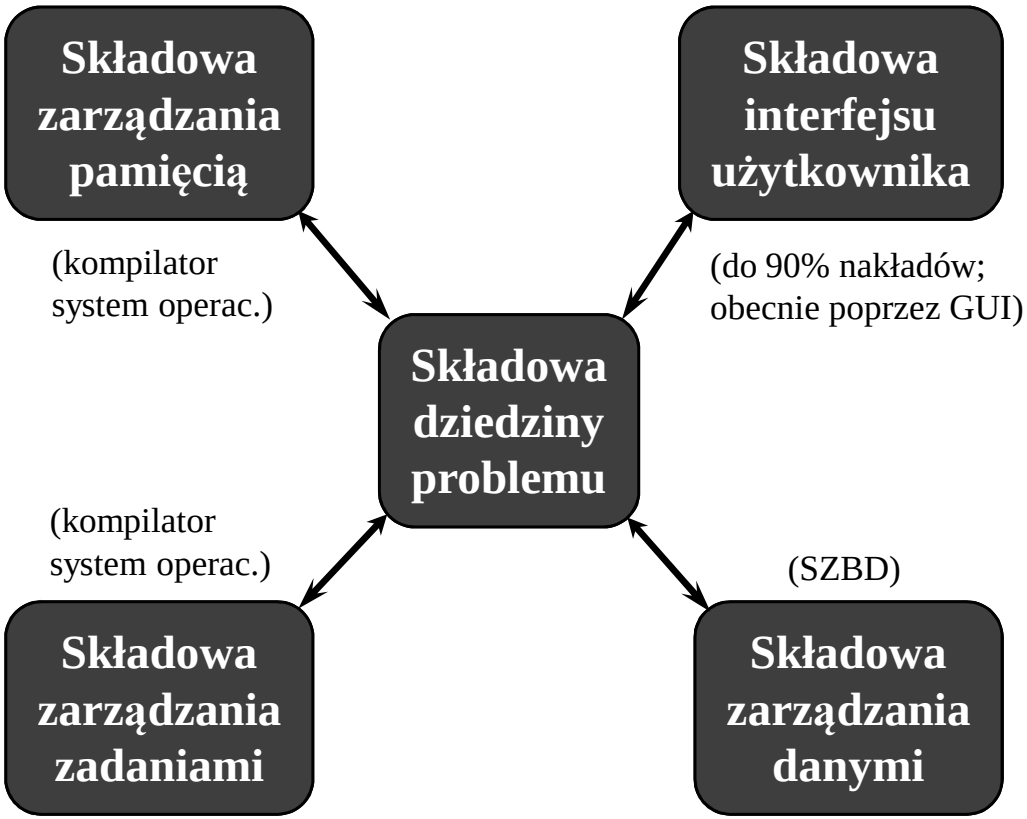
- Opisują zachowanie systemu w reakcji na wewnętrzne lub zewnętrzne zdarzenia.
- Pokazują odpowiedzi systemu na określone stymulacje, dlatego często wykorzystywane są do modelowania systemów czasu rzeczywistego.
- Maszyna stanu pokazuje system w postaci zbioru stanów oraz możliwych przejść pomiędzy nimi wraz ze zdarzeniami, które to przejście powodują.
- Do graficznego przedstawienia maszyny stanów wykorzystuje się diagramy stanów.

Diagramy stanów

Statecharts

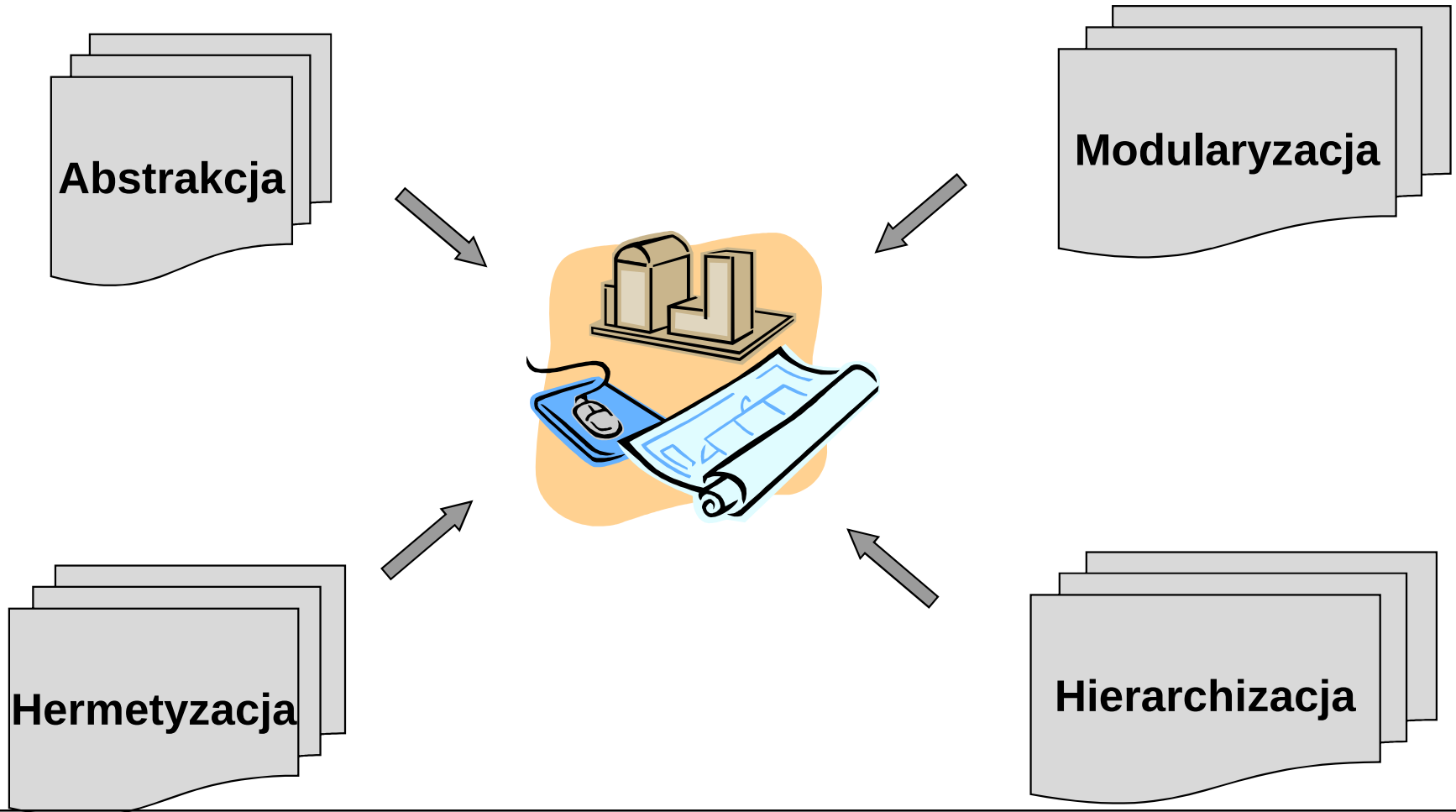
- Umożliwiają poziomowanie modelu (dekompozycję na pod-modele).
- W każdym stanie można opisać akcję która jest wykonywana.
- Mogą być wspomagane tabelami szczegółowo opisującymi stany i pobudzenia.

Składowe systemu w projektowaniu



Obiektowość

- Obiektowość zmniejsza lukę pomiędzy myśleniem o rzeczywistości (dziedzinie problemowej) a myśleniem o danych i procesach, które zachodzą na danych.



Zasada abstrakcji

- Eliminacja lub ukrycie mniej istotnych szczegółów rozważanego przedmiotu lub mniej istotnej informacji.
- Wyodrębnianie cech wspólnych i niezmiennych dla pewnego zbioru bytów i wprowadzanie pojęć lub symboli oznaczających takie cechy.

- Abstrakcja definiuje granice zależne od perspektywy obserwatora.

Przykłady abstrakcji

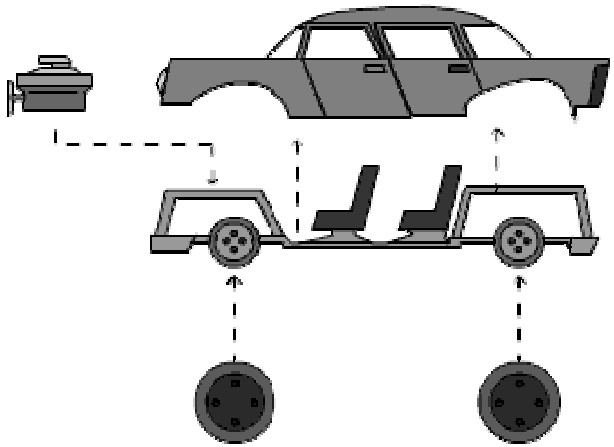
- Student
- Profesor
- Kurs
- Ruchomy pojazd silnikowy, transportujący ludzi z miejsca na miejsce.
- Urządzenie do bezprzewodowego odbioru sygnałów

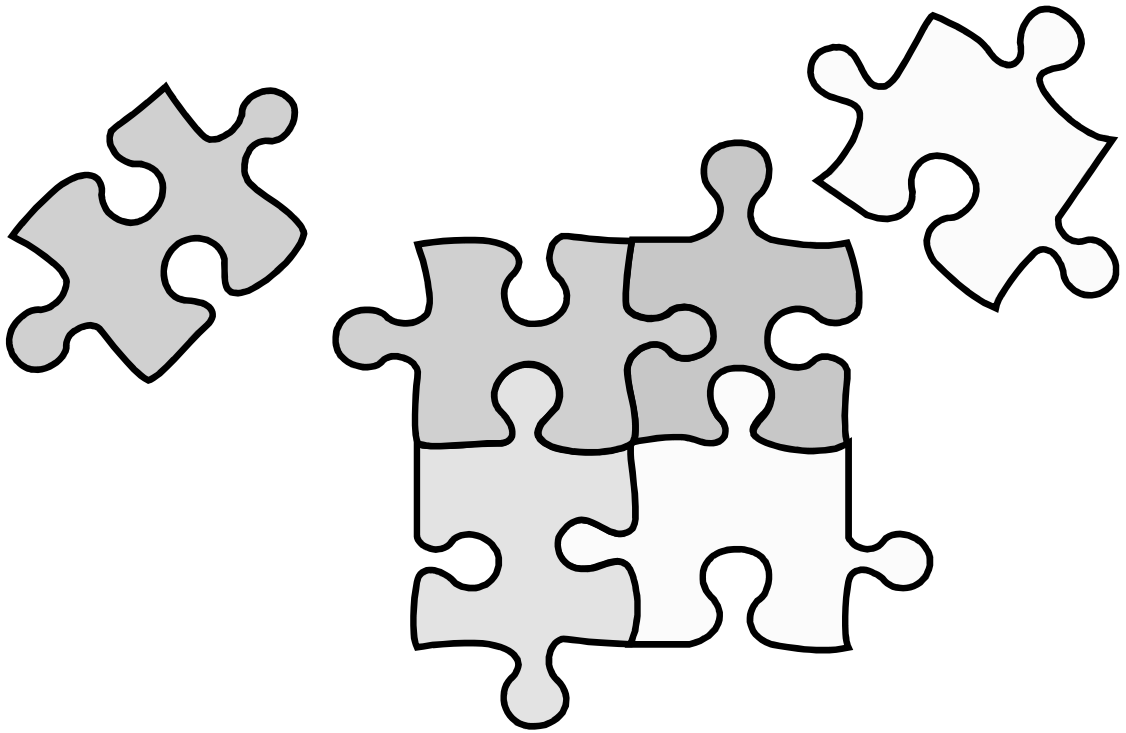
Zasada hermetyzacji – ukrywanie informacji

- Zasada inżynierii oprogramowania (Parnas, 1972): programista ma tyle wiedzieć o obiekcie programistycznym, ile potrzeba, aby go efektywnie użyć. Wszystko, co może być przed nim ukryte, powinno być ukryte.
- Klient zależy od interfejsu.
- Hermetyzacja i ukrywanie informacji jest podstawą pojęć modułu, klasy i ADT.

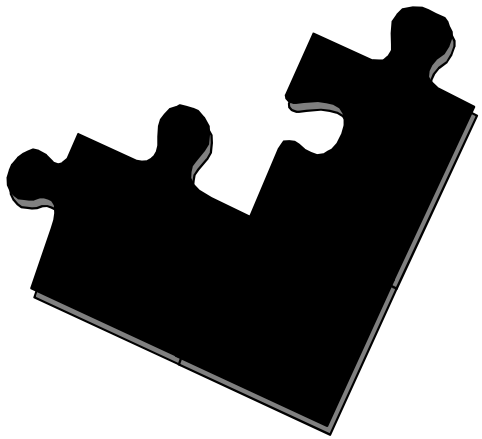
Zasada modularyzacji - dekompozycja

- Rozdzielenie czegoś złożonego na małe łatwiejsze do zarządzania fragmenty.
- Pomaga w zrozumieniu złożonych systemów.

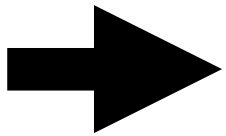




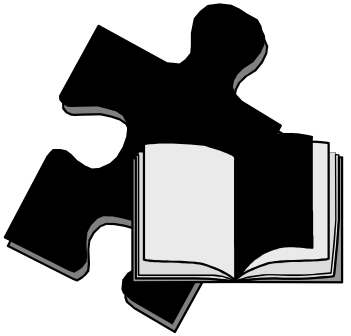
Przykład



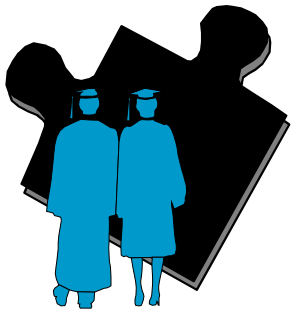
Course Registration System



System Płacowy



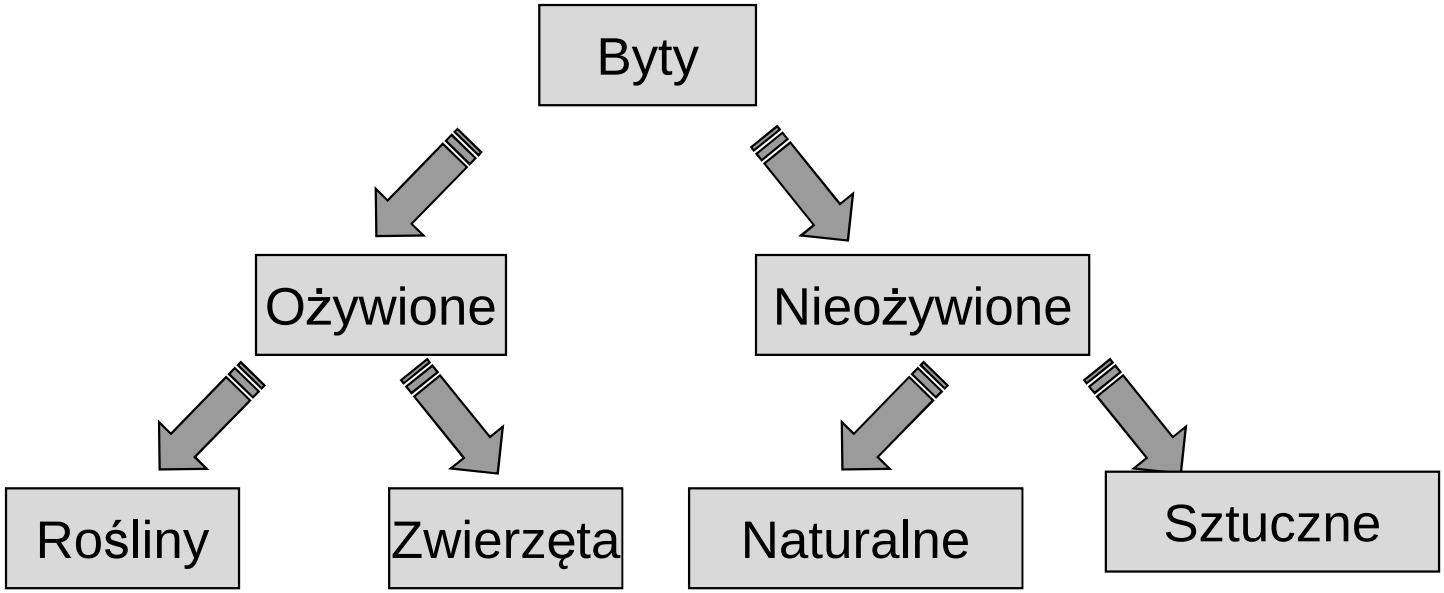
Katalog Kursów



Zarządzanie Studentami

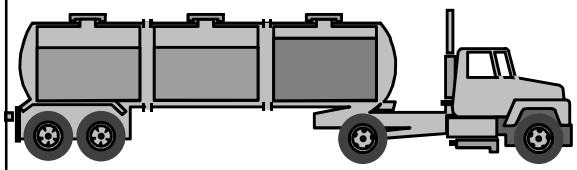
Zasada hierarchizacji

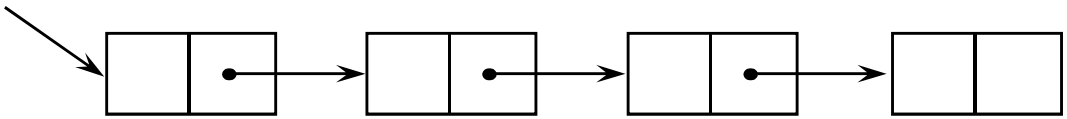
- Porządkowanie (szeregowanie) abstrakcji w strukturę drzewiastą. Rodzaje: hierarcha agregacji, klas, dziedziczenia, typów (Słownik terminów obiektowości, Friesmith, 1995)



Obiekt

- **Nieformalnie**, obiekt jest to byt obserwowalny w rzeczywistości (jej wycinku), koncepcyjny obraz tej rzeczywistości lub jednostka oprogramowania.
- **Formalnie**, obiekt jest jednostką z dobrze zdefiniowanymi granicami, który hermetyzuje stan (*ang. state*) oraz zachowanie (*ang. behavior*).





Podstawowe własności obiektu

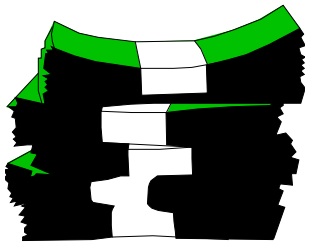
Obiekt jest charakteryzowany poprzez:

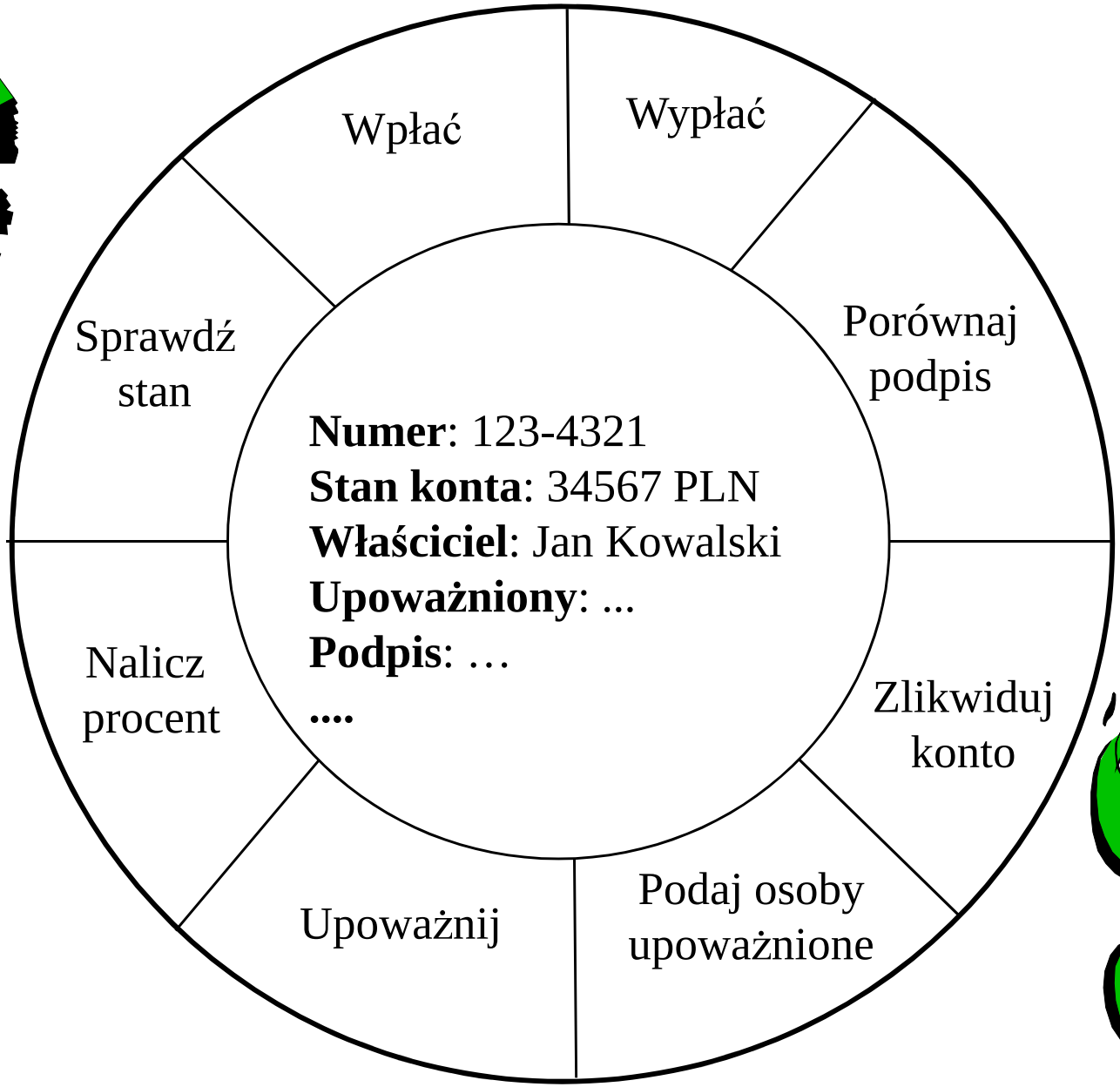
Tożsamość, która odróżnia go od innych obiektów. Tożsamość obiektu jest niezależna zarówno od wartości atrybutów obiektu, jak i od lokacji bytu odwzorowywanego przez obiekt w świecie rzeczywistym czy też lokacji samego obiektu w przestrzeni adresowej komputera.
(W praktyce: tożsamość = trwały **wewnętrzny identyfikator** obiektu)

Stan, który może zmieniać się w czasie (bez zmiany tożsamości obiektu). Stan obiektu w danym momencie jest określony przez aktualne wartości jego atrybutów i powiązań z innymi obiektami.

Obiekt ma przypisane **zachowanie**, tj. zestaw operacji, które wolno stosować do danego obiektu.

Przykład obiektu





Obiekt
KONTO

Modele obiektowe

Object models

- Naturalnie odzwierciedlają elementy rzeczywistości, którymi manipuluje system.
- Opisują system w terminach klas obiektów i relacji pomiędzy nimi.
- Obiekty mogą być rzeczywiste lub abstrakcyjne.
- Identyfikacja klas obiektów jest uważana za trudny proces (wymaga głębokie znajomości dziedziny aplikacji).
- Klasy opisujące obiekty z dziedziny problemowej mają duży potencjał ponownego użycia.

Tworzenie oprogramowania zorientowane obiektowo

- Obiektowa analiza (OOA), projektowanie (OOD) i programowanie (OOP) są ze sobą powiązane ale odrębne.
- **OOA** tworzenie modelu obiektowego dla dziedziny problemu.
- **OOD** – tworzenie systemu zorientowanego obiektowo w celu implementacji wymagań
- **OOP** – realizacja OOD z wykorzystaniem obiektowego języka programowania (Java, C#, etc.)

Charakterystyka OOD

- Obiekty reprezentują elementy rzeczywistości oraz części systemu. Są samo-zarządzalne.
- Obiekty są niezależne, hermetyzują stan i posiadają zachowanie.
- Działanie systemu jest wyrażone w terminach współpracujących ze sobą obiektów.
- Nie istnieją współdzielona dane. Obiekty komunikują się poprzez przesyłanie komunikatów.
- Obiekty mogą być rozproszone oraz mogą działać sekwencyjne lub współbieżnie.

Charakterystyka OOD

- Obiekty reprezentują elementy rzeczywistości oraz części systemu. Są samo-zarządzalne.
- Obiekty są niezależne, hermetyzują stan i posiadają zachowanie.
- Działanie systemu jest wyrażone w terminach współpracujących ze sobą obiektów.
- Nie istnieją współdzielona dane. Obiekty komunikują się poprzez przesyłanie komunikatów.
- Obiekty mogą być rozproszone oraz mogą działać sekwencyjne lub współbieżnie.

Zalety OOD

- Łatwiejsze zarządzanie i konserwacja. Obiekty mogą być postrzegane jako odrębne elementy.
- Obiekty są potencjalnie komponentami ponownego użycia.
- Dla niektórych systemów może istnieć oczywista zależność pomiędzy elementami świata rzeczywistego a obiektami systemu.

Komunikacja obiektów

- Koncepcyjnie obiekty porozumiewają się za pomocą mechanizmu przekazywania komunikatów (ang. *message passing*)
- Komunikat
  - Nazwa usługi, której wymaga wywołujący obiekt
  - Informacje potrzebne usłudze do działania oraz miejsce na przechowanie rezultatu usługi
- Najczęściej komunikat jest oczywiście wywołaniem procedury (lub procedury funkcyjnej)
  - Nazwa usługi – nazwa procedury
  - Informacje – lista parametrów

Komunikat: przykład

// Wywołanie operacji obiektu bufora
// która zwraca następną wartość w buforze

v = bufor.Get () ;

// Wywołanie operacji obiektu reprezentującego
// termostat ustawiająca temperaturę

termostat.setTemp (20);

Serwery i aktywne obiekty

- **Serwer**
  - Obiekt reprezentuje równoległy proces (serwer) z wejściami odpowiadającymi operacjom obiektu. Jeżeli w danym momencie nie wywoływana jest żadna operacja obiekt jest w stanie uśpienia oczekując na zgłoszenia
- **Obiekt aktywny**
  - Obiekty są implementowane jako osobne procesy a wewnętrzny stan obiektu może zostać zmieniony nie tylko przez zewnętrzne odwołania ale również wewnętrznie przez sam obiekt.

Wątek programu w języku Java

- Wątki w Javie są prostymi konstrukcjami umożliwiającymi implementację współbieżnych obiektów.
- Obiekt – wątek musi zawierać operację o nazwie run() uruchamianą automatycznie przez maszynę wirtualną Java.
- Obiekty aktywne zawierają zazwyczaj nieskończoną pętlę – reprezentującą ciągłe przetwarzanie.

Obiekt transpondera satelitarnego w Java

```
class Transponder extends Thread {

        Position currentPosition ;
        Coords c1, c2 ;
        Satellite sat1, sat2 ;
        Navigator theNavigator ;

        public Position givePosition ()
        {
                return currentPosition ;
        }

        public void run ()
        {
                while (true)
                {
                        c1 = sat1.position () ;
                        c2 = sat2.position () ;
                        currentPosition = theNavigator.compute (c1, c2) ;
                }

        }

} //Transponder
```

Proces OOD

- Zrozumienie i zdefiniowanie kontekstu oraz modelu systemu
- Zaprojektowanie architektury systemu
- Identyfikacja głównych obiektów systemu
- Opracowanie modeli projektowych
- Wyspecyfikowanie interfejsów obiektów.

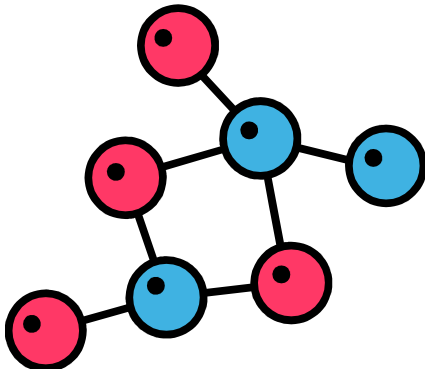
Czynności te przeplatają się i wpływają na siebie. Projekt powstaje przez proponowanie rozwiązań i udoskonalenia w miarę zdobywania nowych informacji

Identyfikacja obiektów

- Wynajdywanie klas obiektów
  - Wykorzystanie analizy gramatycznej opisu systemu w języku naturalnym
  - Wykorzystanie elementów z dziedziny zastosowania
    - rzeczy (np. samolot), ról (np. kierownik), zdarzeń (np. żądanie przelewu), interakcji (np. spotkanie), miejsc (np. biuro), jednostek organizacyjnych, itp.
  - Wykorzystanie podejścia czynnościowego
    - Najpierw zachowanie systemu a potem podejmowanie decyzji kto (co) bierze udział w wykonaniu zadań.
  - Wykorzystanie analizy scenariuszy
    - Rozpoznanie i analiza scenariuszy użycia systemu oraz potrzebnych do ich realizacji obiektów, ich atrybutów i operacji (z wykorzystaniem np. metody kart CRC).

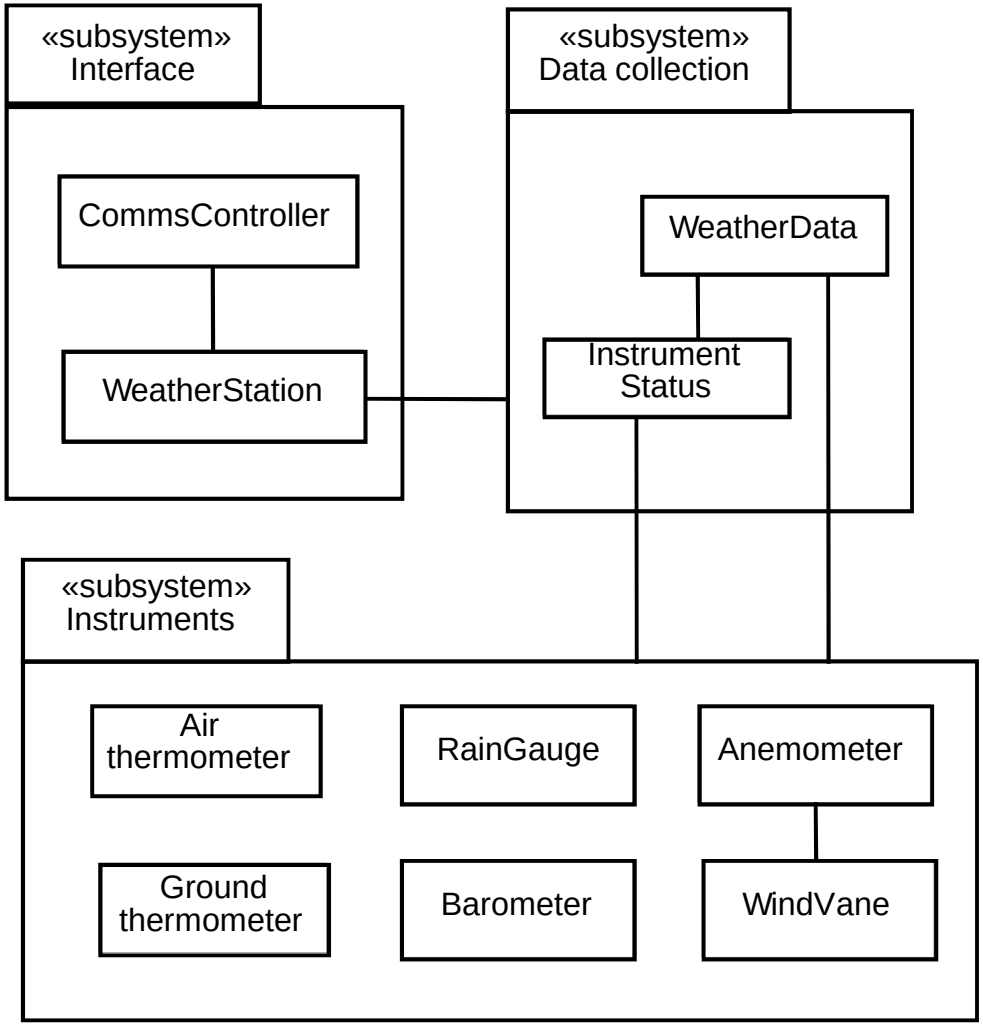
Modele projektowe

- Pokazują obiekty, klasy obiektów i powiązania pomiędzy elementami.
- Modele statyczne
  - Opisują statyczną strukturę systemu w terminach klas obiektów i powiązań pomiędzy nimi
- Modele dynamiczne
  - Opisują interakcje pomiędzy obiektami.

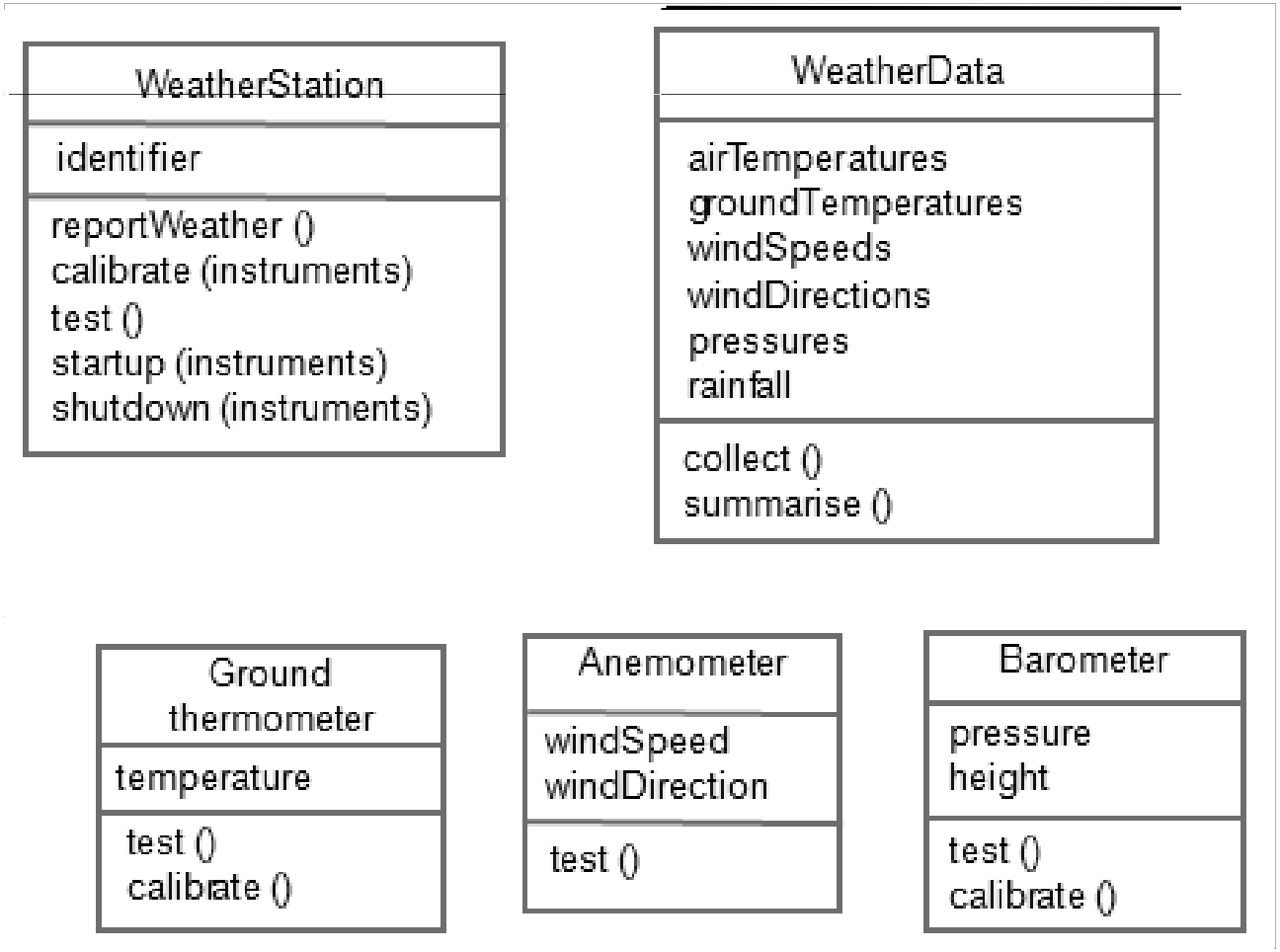


Przykłady modeli projektowych (1)

- **Model podsystemów**
  - logiczne grupowanie obiektów w podsystemy

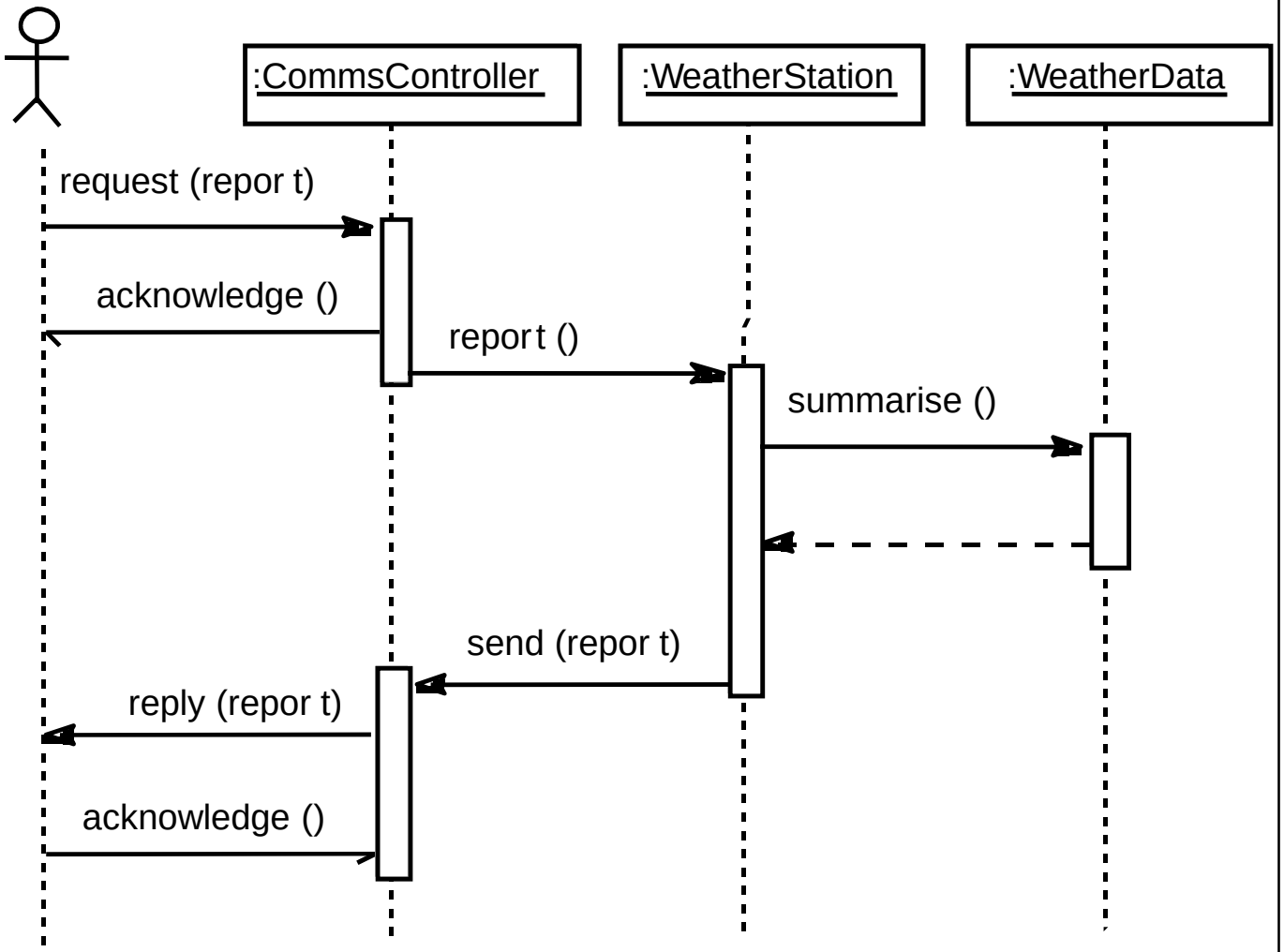


Przykłady modeli projektowych (2)



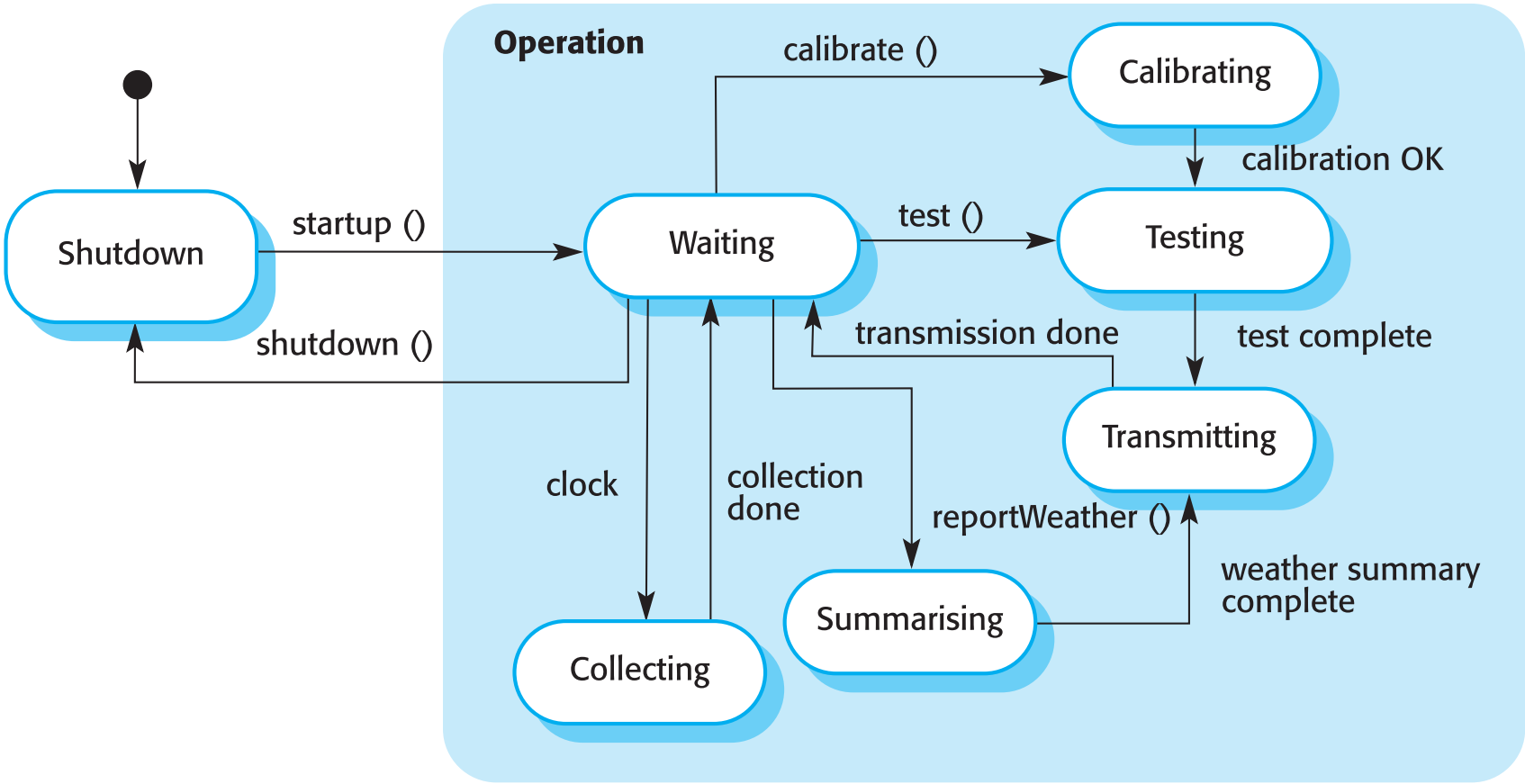
Przykłady modeli projektowych (3)

- **Model dynamiczny**
  - Sekwencja interakcji pomiędzy obiektami



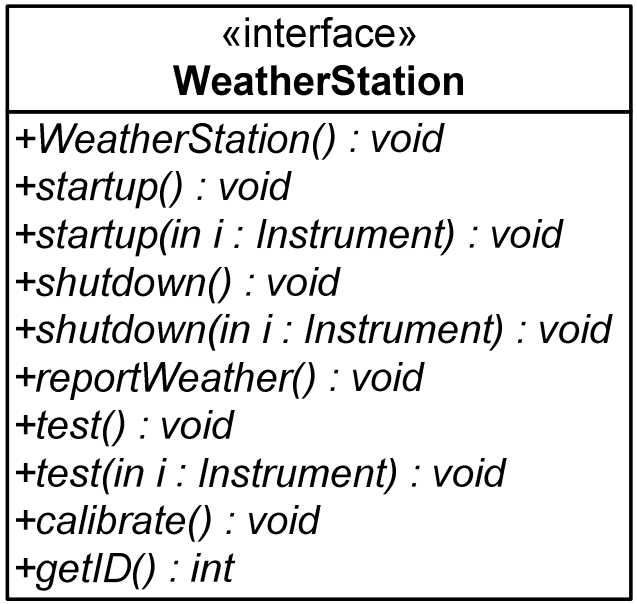
Przykłady modeli projektowych (4)

- **Maszyna stanów**
  - Jak pojedyncze obiekty zmieniają swój stan w odpowiedzi na zdarzenia



Przykłady modeli projektowych (4)

```
interface WeatherStation {

        public void WeatherStation () ;

        public void startup () ;
        public void startup (Instrument i) ;

        public void shutdown () ;
        public void shutdown (Instrument i) ;

        public void reportWeather ( ) ;

        public void test () ;
        public void test ( Instrument i ) ;

        public void calibrate ( Instrument i) ;

        public int getID () ;

} //WeatherStation
```



Powtórne użycie

- W większości dyscyplin inżynieryjnych proces projektowania oparty jest o komponenty ponownego użycia
- W przypadku inżynierii oprogramowania potrzebne jest podobne podejście
- Problemem jest to, że aby można było wielokrotnie użyć komponent oprogramowania trzeba to uwzględnić w fazie jego projektowania

Granulowość ponownego użycia

- Użycie wielokrotne systemów
  - cały system można ponownie użyć poprzez włączenie go do innych systemów (tzw. COTS)
- Użycie wielokrotne komponentów
  - Wielokrotne użycie komponentów systemu (całych podsystemów, modułów, pojedynczych obiektów)
- Wielokrotne użycie funkcji
  - Biblioteki funkcji, powszechnie stosowane od ponad 40 lat.

Korzyści z ponownego użycia

- Zwiększona niezawodność
  - Wykorzystujemy komponenty sprawdzone w działających systemach
  - Redukcja ryzyka w procesie wytwarzania
    - Zwiększanie wiarygodności oszacowania kosztów przedsięwzięcia
  - Wykorzystanie wiedzy specjalistów
  - Zgodność ze standardami
    - Standardy mogą być zaimplementowane jako zbiory komponentów.
  - Przyspieszenie procesu wytwórczego
    - Minimalizacja czasu potrzebnego na tworzenie i testowanie.

Problemy z ponownym użyciem

- Zwiększone koszty pielęgnacji oprogramowania
  - Brak dostępu do kodu źródłowego (alternatywa: OpenSource).
- Brak wspomagania narzędziowego
- Syndrom „nie wymyślono tutaj”
  - Brak zaufania, ja to zrobię lepiej, większym wyzwaniem jest napisanie czegoś od początku
- Prowadzenie biblioteki komponentów
  - Może być kosztowne
- Adaptowanie komponentów
  - Komponenty muszą być najpierw zrozumiane i często zaadaptowane do pracy w nowym środowisku

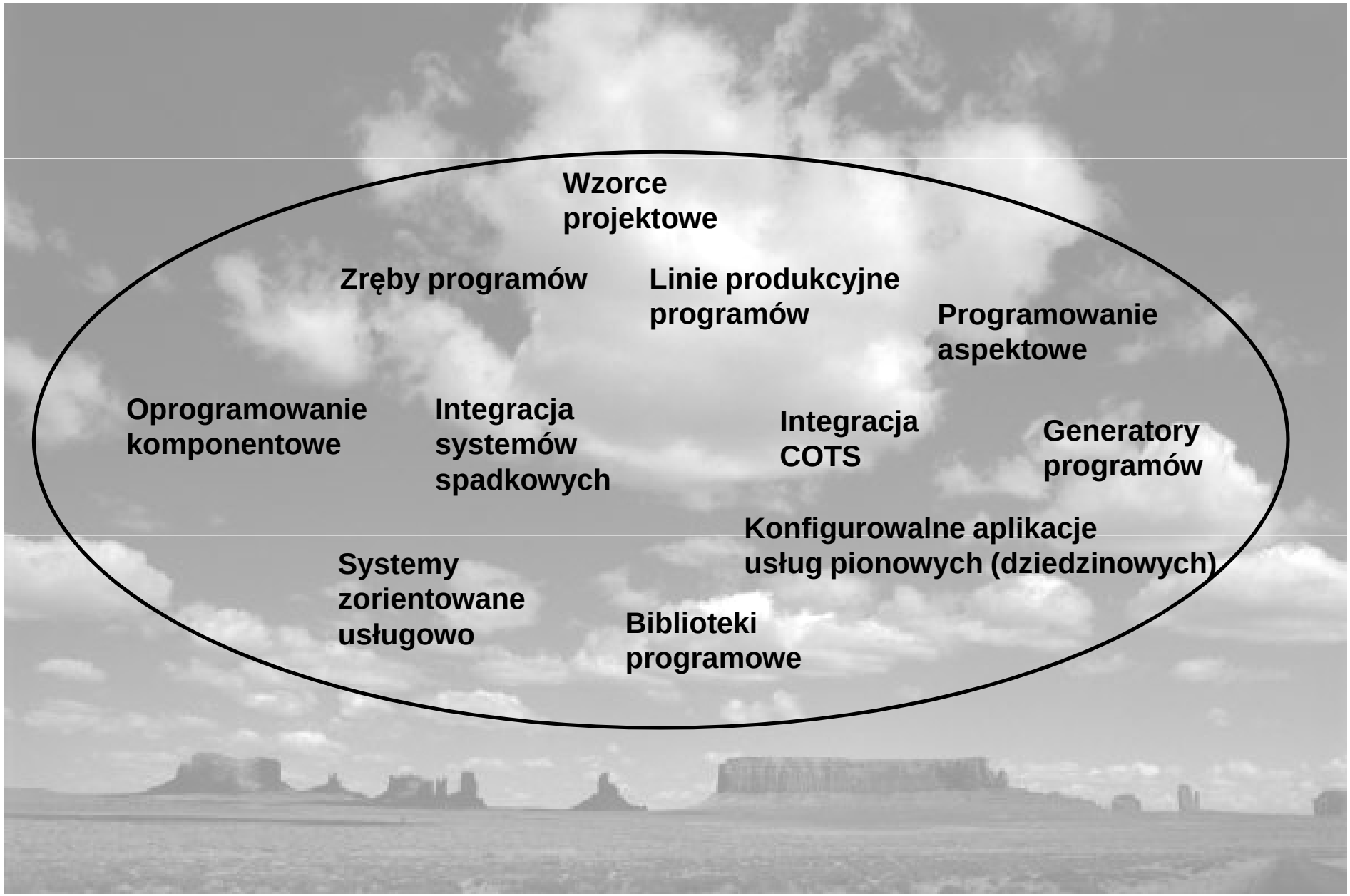
Krajobraz ponownego użycia (1)

- Powtórne użycie jest zazwyczaj postrzegane jako proste wykorzystanie istniejącego komponentu w nowym programie (poczynając od funkcji a na całej aplikacji kończąc).

- Tak naprawdę istnieje wiele różnych obszarów i sposobów ponownego użycia (np. powtórne wykorzystanie pomysłu).

- Pejzaż ponownego użycia jest obecnie bardzo zróżnicowany.

Krajobraz ponownego użycia (2)



Podejścia do ponownego użycia (1)

- **Wzorce projektowe** (ang. *design patterns*):
  - ogólne abstrakcje rozwiązujące problemy pojawiające się w wielu aplikacjach (zestawy obiektów i interakcji)
- **Oprogramowanie komponentowe** (ang. *Component-based development*):
  - Tworzenie systemu na zasadzie integracji komponentów zgodnym ze standardem modelu komponentowego.
- **Zręby aplikacji** (ang. *application frameworks*)
  - Kolekcja klas obiektów, które mogą być wykorzystane i/lub rozbudowane dla utworzenia konkretnej aplikacji

Podejścia do ponownego użycia (2)

- **Integracja systemów spadkowych** (ang. *Legacy systems wrapping*):
  - Integracja istniejących systemów poprzez zdefiniowanie zestawu interfejsów i udostępnienie ich funkcjonalności za pomocą tych interfejsów.
- **Systemy zorientowane usługowo** (ang. *Service-oriented systems*):
  - Tworzenie systemu na zasadzie łączenia współdzielonych usług, które mogą być dostarczane zewnętrznie.
- **Linie produkcyjne programów** (ang. *application product lines*)
  - Typ aplikacji jest uogólniony na zasadzie wspólnej architektury. Istnieje możliwość adaptacji tej architektury na różne sposoby dla różnych klientów.

Podejścia do ponownego użycia (3)

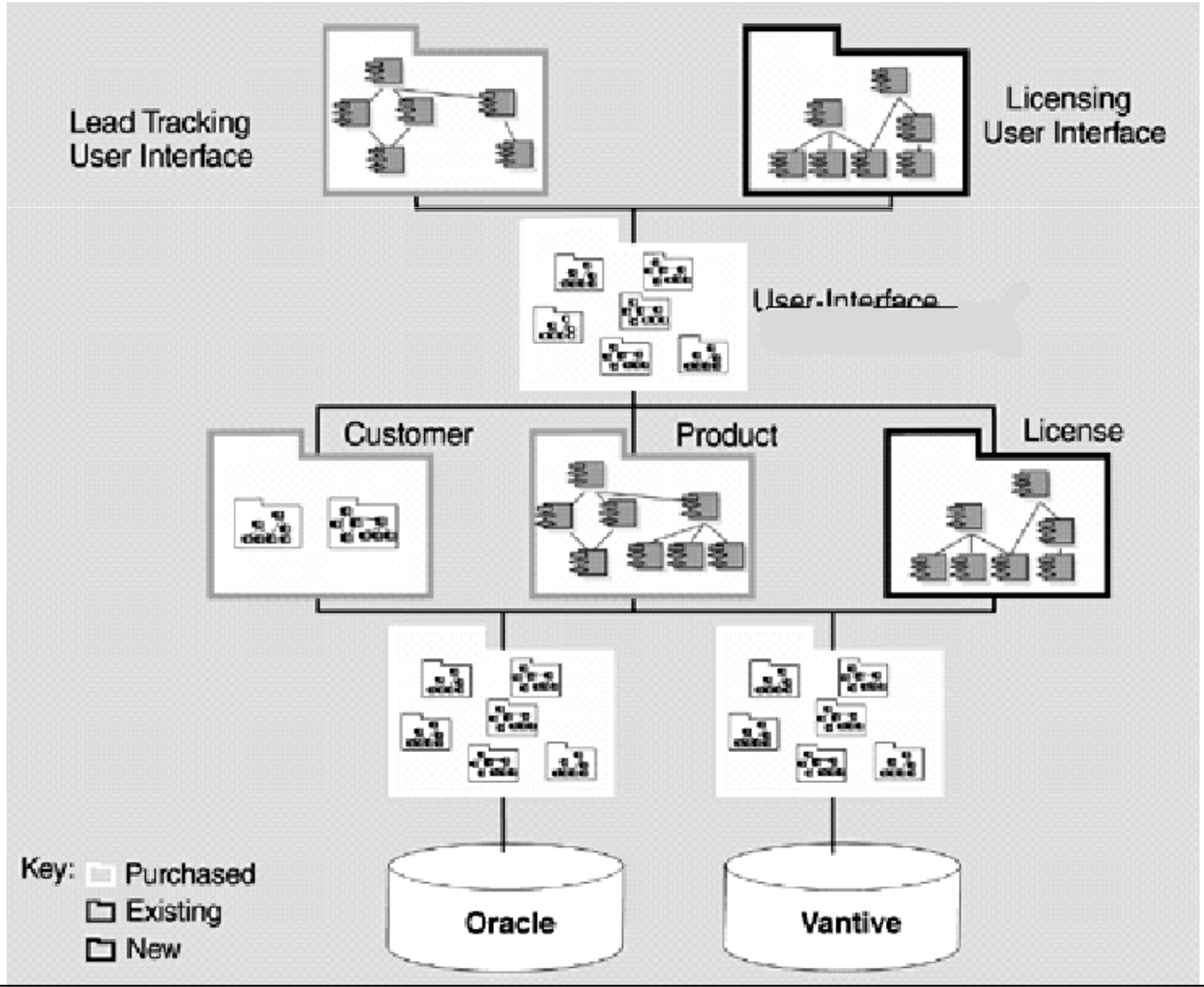
- **Integracja COTS** (ang. *COTS integration*):
  - Tworzenie systemów na zasadzie integracji istniejących aplikacji.
- **Konfigurowalne aplikacje usług pionowych (dziedzinowych)** (ang. *Configurable vertical applications*):
  - Projektuje się generyczny system w taki sposób, aby mógł być łatwo dopasowany do specyficznych wymagań klientów.
- **Biblioteki programowe** (ang. *program libraries*)
  - Biblioteki funkcji i klas implementujące często używane abstrakcje programistyczne.

Podejścia do ponownego użycia (4)

- **Generatory programów** (ang. *Program generators*):
  - Wyspecjalizowane generatory wyposażone w wiedzę o określonych typach aplikacji, które mogą generować całe programy lub ich fragmenty.

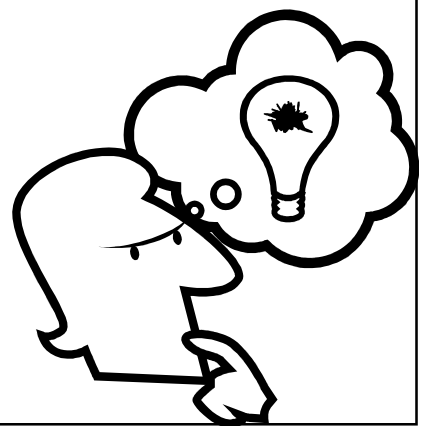
- **Programowanie aspektowe** (ang. *Aspect-oriented software development*):
  - Współdzielone komponenty są automatycznie wplatane w różne miejsca aplikacji podczas procesu kompilacji.

Oprogramowanie komponentowe

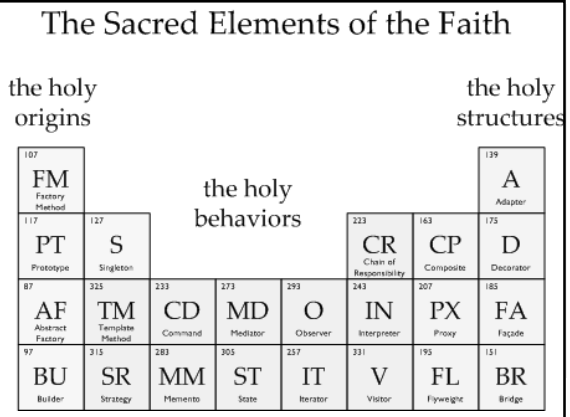


Ponowne użycie ... idei

- Kiedy wykorzystujemy powtórnie program lub komponent projektowy musimy podążać za decyzjami projektowymi podjętymi przez twórcę komponentu
- Może to ograniczać potencjał ponownego użycia
- Może jednak istnieć bardziej abstrakcyjna forma ponownego użycia – wykorzystanie idei/pomysłu rozwiązania konkretnego problemu.
- Dwa główne podejścia to:
  - Wzorce projektowe
  - Programowanie za pomocą generatorów



Wzorce projektowe

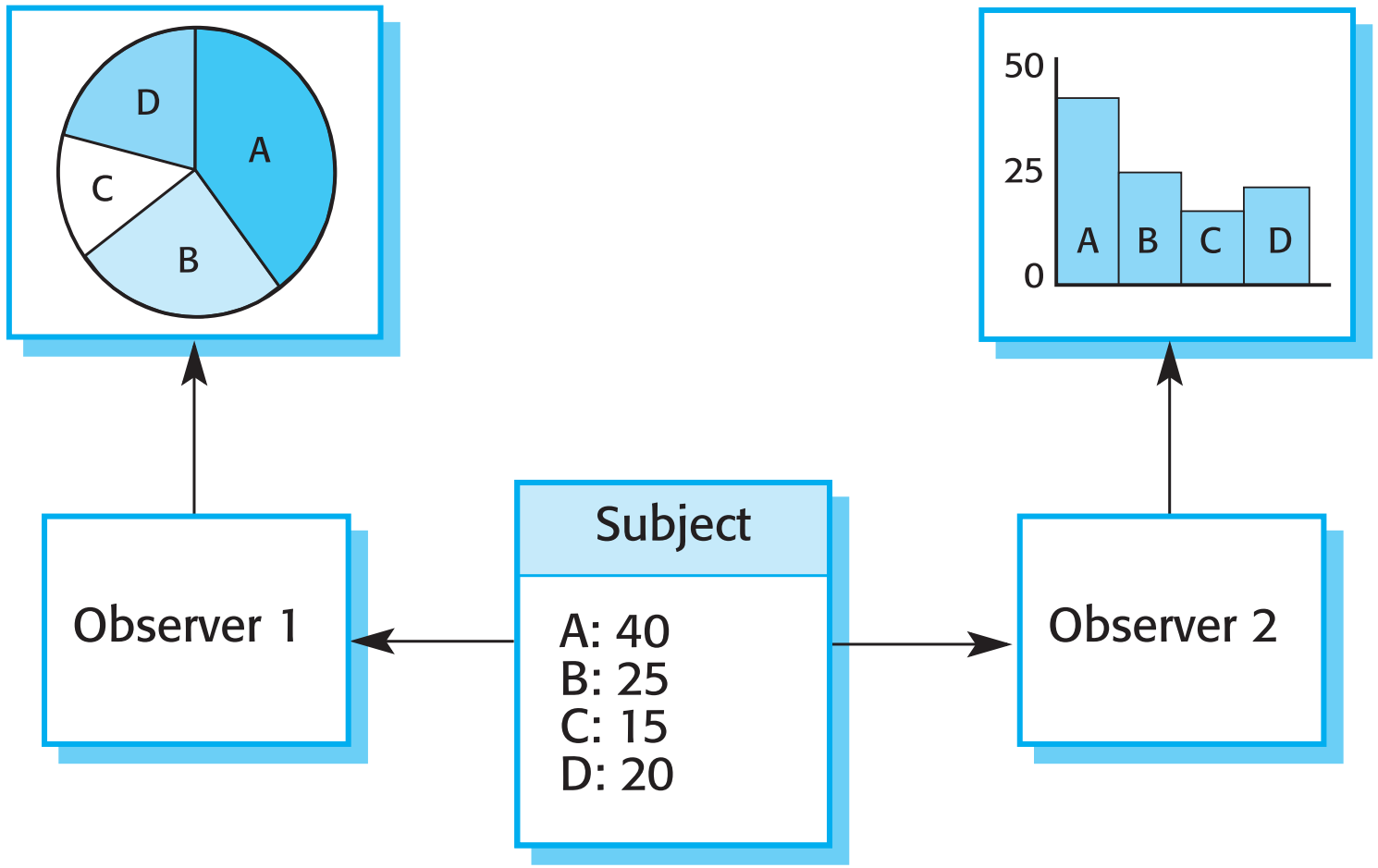


- Mechanizm umożliwiający ponowne użycie wiedzy nt. problemu i sposobu jego rozwiązania.
- Wzorzec jest opisem problemu i istoty jego rozwiązania.
- Powinien być odpowiednio abstrakcyjny, tak aby mógł być wykorzystany w różnorakich konfiguracjach.
- Wzorce bardzo często wykorzystują własności obiektowości, takie jak dziedziczenie czy polimorfizm.

Elementy wzorca

- Nazwa
  - Odzwierciedlająca jego własności (np.. Sigleton, Observer, Template method, Factory method)
- Opis problemu
- Opis rozwiązania
  - Szablon projektu, który może być wykorzystany w różnych konfiguracjach
- Konsekwencje
  - Rezultaty i kompromisy powiązane ze stosowaniem wzorca.

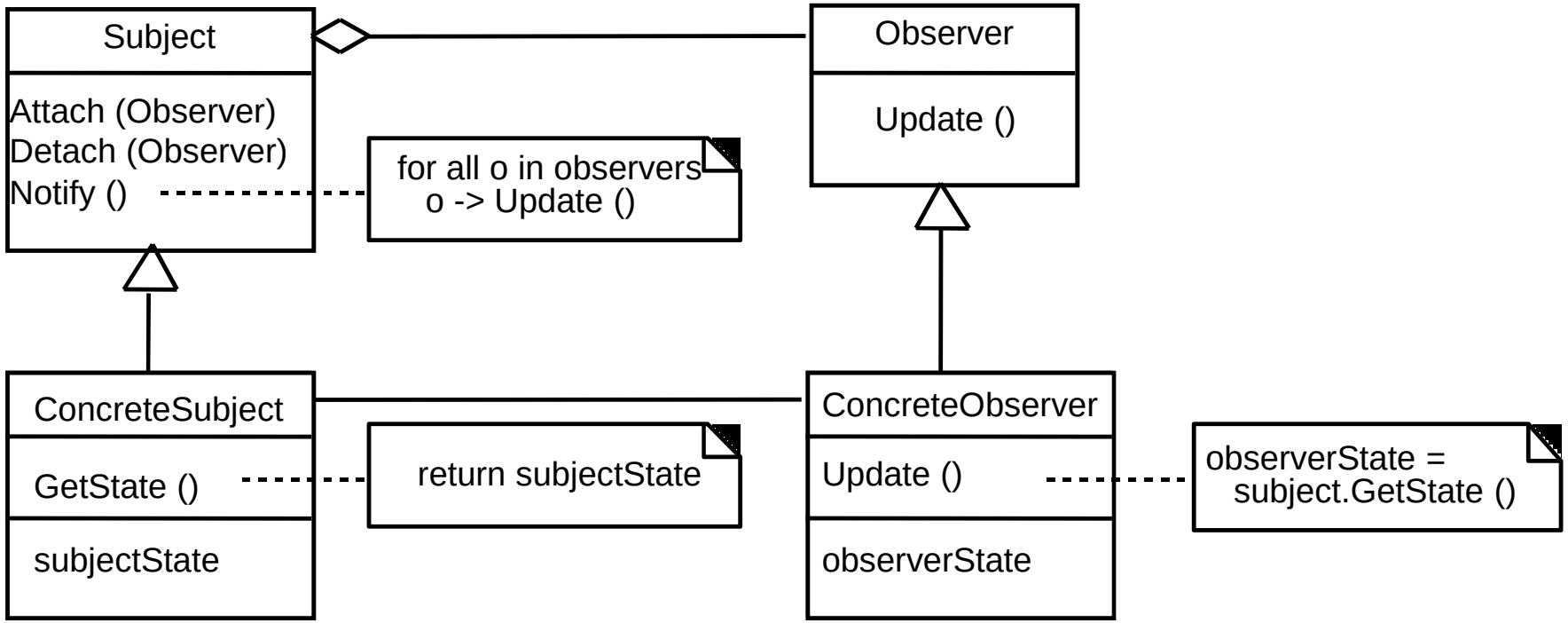
Przykład: Wiele widoków danych



Wzorzec *Observer*

- Nazwa
  - Observer
- Opis
  - Separuje sposób prezentacji stanu obiektu od samego obiektu
- Opis problemu
  - Wykorzystywany, gdy potrzebne są różne sposoby wyświetlania danych obiektu.
- Opis rozwiązania
  - Diagram UML
- Konsekwencje

Wzorzec Observer



Powtórne użycie oparte na generatorach

- Generatory umożliwiają powtórne użycie standardowych wzorców i algorytmów.
- Wzorce i algorytmy są osadzone w generatorze i parametryzowane przez użytkownika. Następnie program jest automatycznie generowany.
- Zazwyczaj wykorzystuje specjalizowane języki dziedzinowe.

Rodzaje generatorów

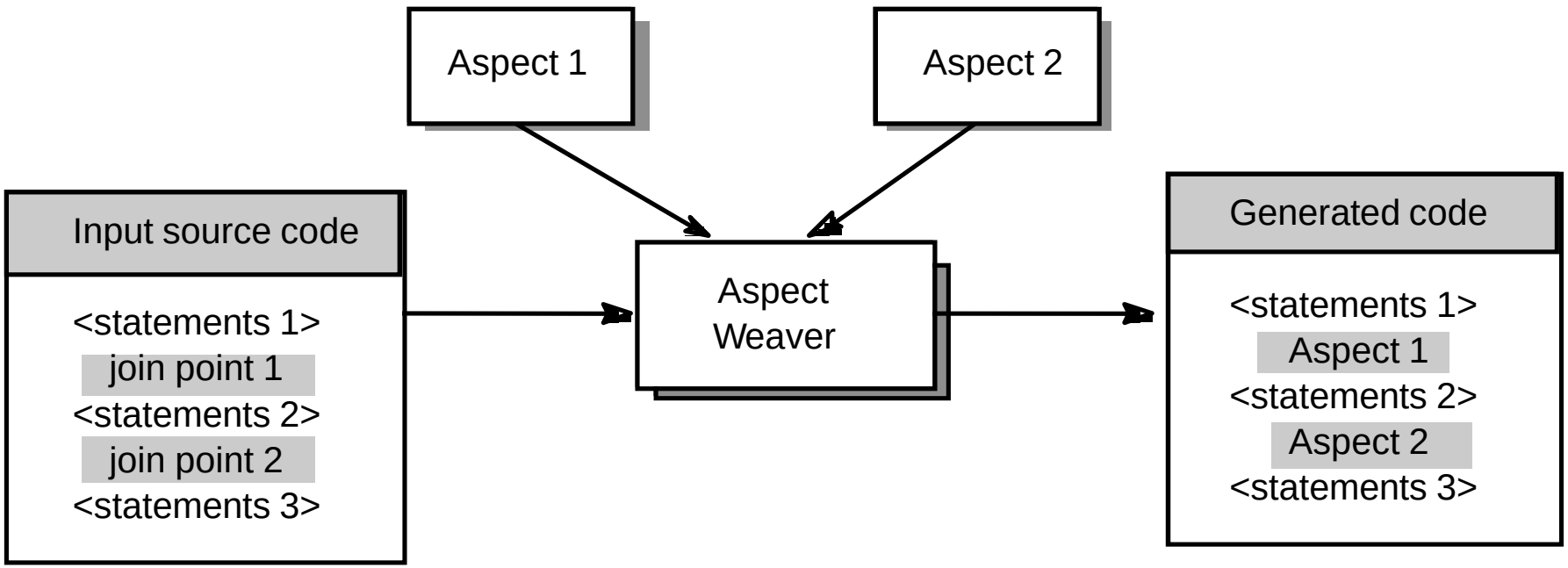
- Generatory aplikacji do przetwarzania danych biznesowych
- Parsery i analizatory leksykalne dla przetwarzania języków (np. dla kompilatorów)
- Generatory kodu w narzędziach CASE.

- Takie generatory są bardzo efektywne z punktu widzenia kosztów jednak ich użycie jest ograniczone do małej liczby dziedzin zastosowań.

Programowanie aspektowe

- Jest związane z podstawowym problemem inżynierii oprogramowania
  - Separation of concerns (separacja spraw)
- Pewne własności oprogramowania nie dają się zdekomponować do pojedynczych jednostek funkcjonalności.
  - Wszystkie moduły muszą implementować bezpieczeństwo
  - Wszystkie moduły muszą implementować monitorowanie
- Te własności przecinają (ang. cross-cut) model komponentów funkcjonalnych

Programowanie aspektowe



Zręby aplikacji

- Projekty podsystemów złożone z kolekcji klas (abstrakcyjnych i konkretnych) oraz interfejsów.
- Implementacja podsystemu odbywa się poprzez dodawania komponentów uzupełniających zrąb (tworzenie klas konkretnych na podstawie klas abstrakcyjnych, definiowanie elementów implementujących interfejsy, itd.).
- Zręby należą do średniej wielkości elementów ponownego użycia.

Klasyfikacja zrębów

- **Zręby infrastruktury systemowej** (ang. *System infrastructure frameworks*)
  - Wspierają tworzenie infrastruktury systemowej (komunikacja, interfejs użytkownika - MVC, kompilatory)
- **Zręby integracyjne warstwy pośredniej** (ang. *Middleware integration frameworks*)
  - Standardy wspierające komunikacje pomiędzy komponentami, wymianę informacji (np. Java RMI, Java Beans, JDBC, ODBC, .NET Remoting, ADO, JSP, ASP)
- **Zręby aplikacji przemysłowych** (ang. *Enterprise application frameworks*)
  - Wspierają wytwarzanie specyficznych typów aplikacji (np. telekomunikacyjnych, systemów finansowych, itp.)

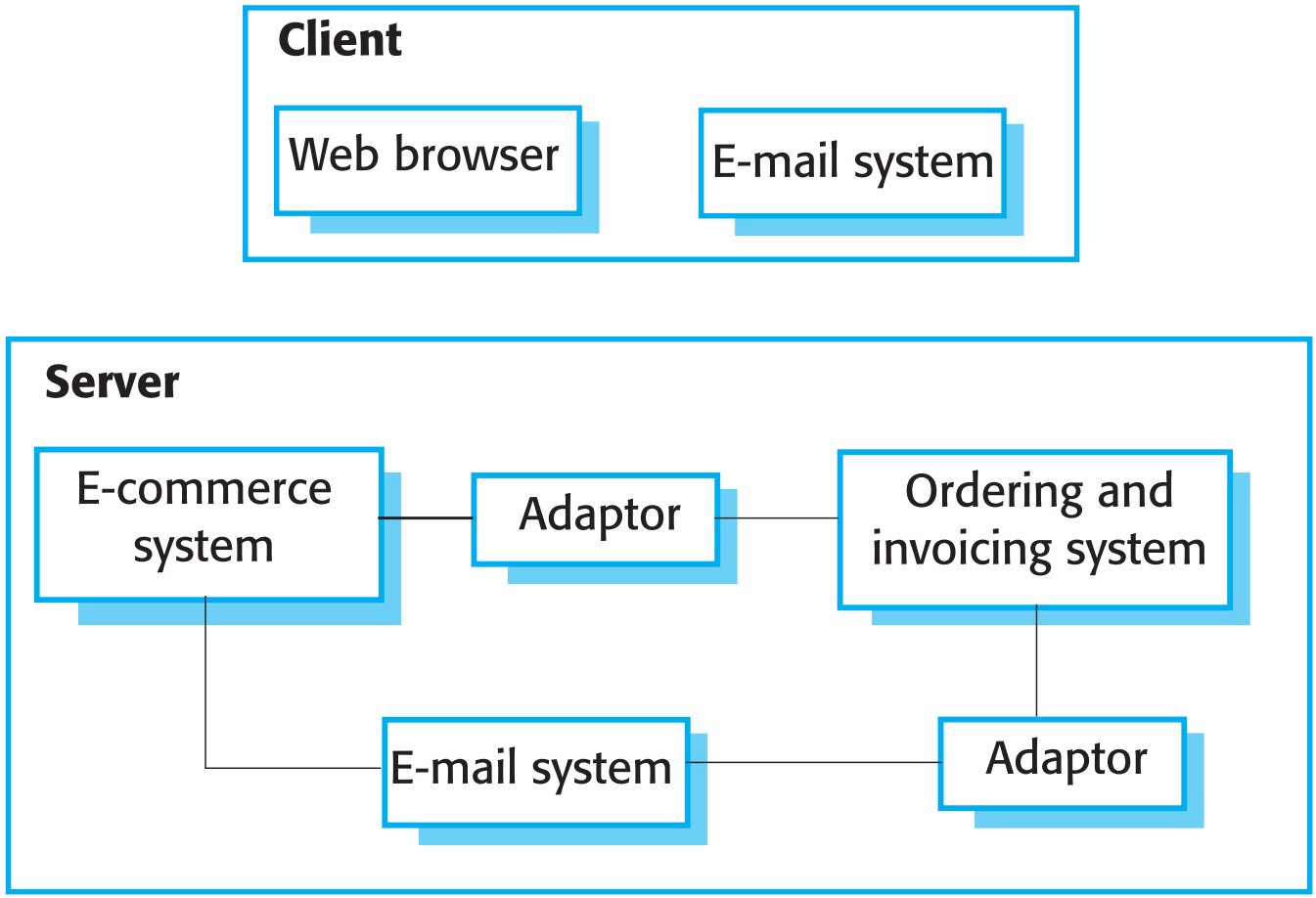
Powtórne użycie aplikacji

- Zakłada powtórne wykorzystanie całych aplikacji poprzez:
  - Konfigurację istniejące aplikacji do nowego środowiska
  - Integrację dwóch lub większej liczby systemów w nowy system.
- Może być dokonywane m.in. poprzez:
  - Integrację COTS
  - Linie produkcyjne programów

COTS

- COTS – Computing/Commercial Off-The-Shelf systems – programowanie „z półki”.
- Systemy COTS są zazwyczaj kompletnymi aplikacjami, które oferują określone API (Application Programming Interface).
- Tworzenie dużych systemów poprzez integrację COTS jest opłacalne dla pewnych typów systemów (np. E-commerce).
- Zalety: szybsze tworzenie i zazwyczaj niższe koszty.

System E-zaopatrzenie



Problemy COTS

- Brak kontroli nad wydajnością i funkcjonalnością
  - Systemy COTS mogą być mniej efektywne niż to się początkowo wydawało
- Problemy z integracją i współpracą
- Brak kontroli nad ewolucją systemu
- Niewielki zakres wsparcia dostawców COTS
  - Ale to się powoli zmienia na lepsze

Do poczytania

- Sommerville I.*: Inżynieria Oprogramowania,* rozdział 12 i 14.

- Wzorce projektowe
  - Gamma i in.: *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku,* WNT, Wa-wa, 2005
  - Metsker S.J.: *C#. Wzorce projektowe*, Helion, Gliwice, 2005.
  - http://home.earthlink.net/~huston2/dp/patterns.html

Do poczytania cd.

- Generatory programów, programowanie aspektowe
  - Czarnecki K., Eisenecker U.W.: *Generative Programming. Methods, Tools and Applications*, Addison-Wesley, 2000.

- Projektowanie obiektowe – roadmap
  - http://www.sei.cmu.edu/str/descriptions/oodesign.html

Model-Widok-Koordynator (*ang. Model-View-Controler*)