Grupa 1

a) Jakie informacje na karcie CRC
    -opis klasy, za co odpowiada, z jakimi klasami współpracuje
b) Czym jest abstrakcja
    -Poziom złozonosci modelu ze swiata rezczywistego
c) Klasa obiektu a Obiekt
    -Klasa - schemat ogólny
    -Obiekt - instancja klasy, pola sa zainicjalizowane
d) roznica pomiedzy kompozycja a agregacja
    -agregacja - obiekt sklada sie z innych obiektow, niezaleznosc obiektu podrzednego od nadrzednego
    -kopmozycja szczegolny przypadek agregacji, istnienie obiektu bez obiektu rodzica nie ma sensu
e) klasa abstrakcyjna Java
    - oznaczona slowem kluczowym abstract
    - nie mozna tworzyc instancji klasy abstrakcyjnej(zbyt ogolna)
    - moze zawierac metody abstrakcyjne
f) zasada DRY
    -dont repeat yourself - unikanie powtarzania tych samych czesci kodu/logiki w roznych miejscach
g) czym jest hermetyzacja
    -ukrywanie szczegolow implementacji
    -ograniczanie dostepu do pol/metod
h) klasa spojna
    -klasa implementuje okreslony waski zakres funkcjonalnosci
i) zasada luznych powiązań
    -przypisywanie odpowiedzialnosci klasie tak zeby liczba powiazan i innymi klasami byla jak najnizsza
j) zasada postawien Liskov
    - klasa dziedziczaca po klasie bazowej nie powienna zmieniac funkcjonalnosci tylko ja rozszezac
k) metoda statyczna java
    -mozna ja wywolac nie tworzac obiektu
    - nie moze sie odwolywac do niestatycznych elementow klasy
l) garbage collector
    -komponet JVM ktorego glownym zadaniem jest usowanie z pamieci nieuzywanych obiektow

Grupa 2

a)informacje na diagramie przypadkow uzycia
    -z pkt widzenia aktora pokazuje co mozna zrobic z systemem
b)interfejs w programowaniu obiektowym
    -lista metod publicznych, zawiera jedynie specyfikacje
c)podstawowe roznice pomiedzy podejsciem procedyralnym a obiektowym
    -Proceduralne - dzielenie kodu na procedury wykonujace scisle okreslone zadanie, nie komunikuja sie ze soba
    -Obiektowe - obiekty w ktorym dane i procedury sa w sobie powiazane , kozysta z obiektow komunikujacych sie ze soba w celu wykonania zadania
d) Polimorfizm
    -mozliwosc zadeklarowania metody o takiej samej nazwie a o innej tresci w kilku klasach
e) Interfejs w Java
    - deklaracje metod
    -mozna implementowac nieskonczenie wiele interfejsow
f) zasada SRP
    -kazda klasa powinna odpowiadac za jak najmniejszy fragment logiki programu
g) zasada segregacji interfejsow
    -klasa udostepnia tylko te interfejsy ktore sa niezbedne do wykonania programu
h) zasada otwarte-zamkniete
    -elementy systemu powinny byc otwarte na rozszezenie ale zamkniete na modyfikacje
i) zasada information expert

-programista powienien delegowac nowa odpowiedzialnosc do klasy zaweirajacej najwiecej info do zrealizowania nowej funkcjonalnosci

j)czy zasada liskov zabrania instanceof

-nie?

k)klasa zadeklarowana jako finalna

-nie mozna po niej dziedziczyc

-metody sa automatycznie finalne

l)konstruktor

-kod uruchamiany w trakcie tworzenia nowego obiektu do jego stworzenia

---

S - Single responsibility principle

- Klasa powinna być odpowiedzialna za pojedynczy obszar. Nie powinno być więcej powodów niż jeden, aby klasa się zmieniła.
- Odpowiedzialność może być zdefiniowana jako powód do zmiany, zatem klasa powinna mieć tylko jeden taki powód, a więc jedną odpowiedzialność.
- Klasa realizuję pojedyncze zadanie.
- Gdy wymagana jest zmiana, powinna ona objąć tylko jedną klasę lub pakiet.

O - Open/closed principle

- Powinieneś być w stanie rozszerzać swoje klasy bez jej modyfikacji.
- Klasy powinny być otwarte na rozszerzanie, a zamknięte na modyfikację.
- Polegaj na abstrakcji i polimorfizmie
- Łatwo złamać tą zasadę, gdy mamy przypadki kontrolowane przez ify czy konstrukcję switch.

L - Liskov substitution principle

- Klasy w programie powinny być podmienialne przez swoje podklasy bez naruszania poprawności programu, czyli klasa dziedzicząca musi być dobrym odpowiednikiem klasy bazowej.
- Podklasa nie powinna robić mniej niż klasa bazowa. Czyli zawsze powinna robić więcej.
- Zobacz popularny przykład złego zastosowania dziedziczenia “Square extends Rectangle”

I - Interface segregation principle

- Wiele mniejszych, konkretnych interfejsów jest lepsze od pojedynczego ogólnego interfejsu.
- Powinno się projektować małe i zwarte interfejsy.
- Klasa nie powinna implementować interfejsu, przez który naruszy Single responsibility principle
- Klasa która implementuje interfejs nie może być zmuszana do implementowania metod, których nie potrzebuje, a tak jest często w przypadku dużych iterfejsów.

D - Dependency inversion principle

- Niskopoziomowe klasy powinny zależeć od klas wysoko poziomowych, a obie od swoich abstrakcji.
- Abstrakcję nie powinny polegać na szczegółach implementacyjnych.
- Szczegóły implementacyjne powinny polegać na abstrakcji.
- Czyli w żadnej definicji funkcji i w żadnej deklaracji zmiennej nie powinniśmy używać nazwy klasy, tylko jej abstrakcji (np. interfejs, klasa abstrakcyjna).

KISS - Keep it simple, stupid!

- Prostota (i unikanie złożoności) powinna być priorytetem podczas programowania. Kod powinien byc łatwy do odczytania i zrozumienia wymagając do tego jak najmniej wysiłku.
- Większość systemów działa najlepiej, gdy polegają na prostocie, a nie na złożoności.
- Staraj się, aby twój kod podczas analizy nie zmuszał do zbytniego myślenia.
- Gdy po jakimś czasie wracasz do swojego kodu i nie wiesz co tam się dzieje, to znak, że musisz nad tym popracować

DRY - Don't Repeat Yourself

- Jedna z podstawowych zasad programowania - nie powtarzaj się. Wielokrotne użycie tego samego kodu to podstawa programowania.
- Jeśli jesteś blisko powtórzenia (np. chcesz zastosować kopiuj/wklej, seria ifów lub w kodzie występują podobne zachowania) pomyśl nad stworzeniem abstrakcji (pętla, wspólny interfejs, funkcja, klasa, jakiś wzorzec projektowy np. Strategia itp.), którą będziesz mógł wielokrotnie wykorzystać.

2.1 Creator

Nazwa	**Creator** (Twórca)
Problem	Kto tworzy instancje klasy A?
Rozwiązanie	Przydziel zobowiązanie tworzenia instancji klasy A klasie B, jeżeli zachodzi jeden z warunków: • B „zawiera" A lub agreguje A (kompozycja) • B zapamiętuje A • B bezpośrednio używa A • B posiada dane inicjalizacyjne dla A

2.2 Information Expert

Nazwa	**Information Expert**
Problem	Jak przydzielać obiektom zobowiązania?
Rozwiązanie	Przydziel zobowiązanie "ekspertowi" – tej klasie, która ma *informacje* konieczne do jego realizacji.

2.4 Low Coupling

Nazwa	**Low Coupling**
Problem	Jak zmniejszyć liczbę zależności I zasięg zmian, a zwiększyć możliwość ponownego wykorzystania kodu?
Rozwiązanie	Przydziel odpowiedzialność tak, aby ograniczyć stopień sprzężenia (liczbę powiązań obiektu). Stosuj tę zasadę na etapie projektowania.

2.5 High Cohesion

Nazwa	**High Cohesion**
Problem	Jak sprawić by obiekty miały jasny cel, były zrozumiałe i łatwe w utrzymaniu?
Rozwiązanie	Przydziel odpowiedzialność by spójność pozostawała wysoka.

Klasa o niskiej spójności wykonuje niepowiązane zadania lub ma ich zbyt dużo:

- Trudno je zrozumieć
- Trudno je ponownie wykorzystać
- Trudno je utrzymywać
- Są podatne na zmiany

2.6 Polymorphism

Nazwa	**Polymorphism**
Problem	Jak obsługiwać warunki zależne od typu?
Rozwiązanie	Przydziel zobowiązania - przy użyciu operacji polimorficznych – typom dla których to zachowanie jest różne

2.9 Protected Variations

Nazwa	**Protected Variations**
Problem	Jak projektować obiekty, by ich zmiennić nie wywierała szkodliwego wpływu na inne elementy?
Rozwiązanie	Rozpoznaj punkty zmienności o otocz je stabilnym interfejsem.

Law of Demeter

Adapter (także: opakowanie, ang. *wrapper*) – strukturalny wzorzec projektowy, którego celem jest umożliwienie współpracy dwóm klasom o niekompatybilnych interfejsach. Adapter przekształca interfejs jednej z klas na interfejs drugiej klasy[1]. Innym zadaniem omawianego wzorca jest opakowanie istniejącego interfejsu w nowy

Rodzaje symulacji

- **Przewidywalność:**
  - **deterministyczne,** pomija się składniki losowe modelu, co- w modelach liniowych- oznacza operowanie wartościami oczekiwanymi poszczególnych zmiennych,
  - **stochastyczne** -uwzględnia się składnik losowy i właściwości jego rozkładu (w program obliczeniowy musi być wtedy wbudowany odpowiedni podprogram generujący realizację składnika losowego i uwzględniający rzeczywiste własności jego rozkładu).
- **Upływ czasu:**
  - **czas ciągły,** czas rośnie ciągłymi przyrostami, a krok czasowy wybiera się optymalnie z powodu "zasobożerności" systemu, jego sprawności i postaci,
  - **czas dyskretny,**
  - **symulacja zdarzeń dyskretnych** czas rośnie stopniowo, lecz jego przyrosty są różne.
- **Sposób symulacji:**
  - **oparty o modele analityczne,**
  - **agent-based.** jest to specjalna forma symulacji dyskretnych, nie opierających się na danym modelu, lecz możliwa do przedstawienia.

Kreacyjne:

- **Singleton** - ogranicza możliwość tworzenia obiektów danej klasy do jednej instancji oraz zapewnia do niej globalny dostęp.
  Singleton jest jednym z najprostszych wzorców projektowych. Jego celem jest ograniczenie możliwości tworzenia obiektów danej klasy do jednej instancji oraz zapewnienie globalnego dostępu do stworzonego obiektu – jest to obiektowa alternatywa dla zmiennych globalnych.

- **Builder** - rozdzielenie sposobu tworzenia obiektów od ich reprezentacji,
  jest wzorcem, gdzie proces tworzenia obiektu podzielony jest na kilka mniejszych etapów, a każdy z nich może być implementowany na wiele sposobów. Dzięki takiemu rozwiązaniu możliwe jest tworzenie różnych reprezentacji obiektów w tym samym procesie konstrukcyjnym.

- **Abstract Factory** -Fabryka abstrakcyjna , polega na dostarczeniu interfejsu do tworzenia różnych obiektów jednego typu (tej samej rodziny) bez specyfikowania ich konkretnych klas,
  Fabryka abstrakcyjna jest wzorcem projektowym, którego zadaniem jest określenie interfejsu do tworzenia różnych obiektów należących do tego samego typu (rodziny). Interfejs ten definiuje grupę metod, za pomocą których tworzone są obiekty.

- **Prototype** - umożliwia tworzenie obiektów danej klasy z wykorzystaniem już istniejącego obiektu, zwanego prototypem,
  Prototyp jest wzorcem, opisującym mechanizm tworzenia nowych obiektów poprzez klonowanie jednego obiektu macierzystego. Mechanizm klonowania wykorzystywany jest wówczas, gdy należy wykreować dużą liczbę obiektów tego samego typu lub istnieje potrzeba tworzenia zbioru obiektów o bardzo podobnych właściwościach.

- **Factory Method** -dostarcza interfejs do tworzenia obiektów nieokreślonych typów, pozwala podklasom zdecydować, jakiego typu będzie obiekt,
  Wzorzec **metody wytwórczej** dostarcza abstrakcji do tworzenia obiektów nieokreślonych, ale powiązanych typów. Umożliwia także dziedziczącym klasom decydowanie jakiego typu ma to być obiekt.

Strukturalne:

- **Wraper** -umożliwia współpracę dwóch klas o niekompatybilnych interfejsach
  Wzorzec **adapter** (znany także pod nazwą **wrapper**) służy do przystosowania interfejsów obiektowych, tak aby możliwa była współpraca obiektów o niezgodnych interfejsach. Szczególnie przydaje się przypadku wykorzystania gotowych bibliotek o interfejsach niezgodnych ze stosowanymi w aplikacji. W świecie rzeczywistym adapter to przejściówka, np. przejściówka do wtyczki gniazdka angielskiego na polskie.

- **Decorator -** pozwala na dynamicznie dodawanie nowych funkcjonalności do istniejących klas podczas działania programu
  Wzorzec projektowy Dekorator pozwala na dynamiczne przydzielanie danemu obiektowi nowych zachowań. Dekoratory dają elastyczność podobną do tej, jaką daje dziedziczenie, oferując jednak w zamian znacznie rozszerzoną funkcjonalność

- **Facade -** pojedyncza klasa, która służy do uproszczenia i uporządkowania możliwości systemu, służy do ułatwienia korzystania z systemu,
  Fasada służy do ujednolicenia dostępu do złożonego systemu poprzez udostępnienie uproszczonego i uporządkowanego interfejsu programistycznego. Fasada zwykle implementowana jest w bardzo prosty sposób – w postaci jednej klasy powiązanej z klasami reprezentującymi system, do którego klient chce uzyskać dostęp.

- **Composite - strukturalny wzorzec projektowy, którego celem jest składanie obiektów w taki sposób, aby klient widział wiele z nich jako pojedynczy obiekt.**
  Composite jest bardzo często stosowanym wzorcem służącym do reprezentacji struktur drzewiastych typu całość-część tak, aby sposób zarządzania strukturą nie zależał od jej złożoności. Jest często stosowany w obiektowych bibliotekach okienkowych jako metoda zarządzania widokami zbudowanymi z wielu widget'ów.

- **Bridge - służy do oddzielenia interfejsu od jego implementacji,**
  Most (*ang. bridge*) - strukturalny wzorzec projektowy. Pozwala na modyfikowanie implementacji oraz abstrakcji w czasie działania programu. Interfejs zostaje całkowicie odizolowany od swojej implementacji. Dzięki temu zyskujemy możliwość oddzielnego modyfikowania abstrakcji oraz oddzielnej modyfikacji implementacji. Przydatny może być w sytuacji, gdy graficzny interfejs użytkownika (GUI) musi wyglądać inaczej w zależności od posiadanego systemu operacyjnego. Zmiany w kodzie mają charakter dynamiczny (wszystkie modyfikacje dokonywane są w trakcie działania programu). Dodatkowo wzorzec ten może służyć do odseparowania klienta od implementacji określonego interfejsu.

- **Flyweight - łuży do zmniejszenia wykorzystania pamięci poprzez obsługę wielu małych obiektów równocześnie, przy pomocy współdzielenia wspólnych cech,**
  strukturalny wzorzec projektowy, którego celem jest zmniejszenie wykorzystania pamięci poprzez poprawę efektywności obsługi dużych obiektów zbudowanych z wielu mniejszych elementów poprzez współdzielenie wspólnych małych elementów[. Należy do grupy wzorców skatalogowanych przez Gang of Four.

- **Proxy -polega na tworzeniu obiektu, który zastępuje inny obiekt.**
  Pełnomocnik jest wzorcem projektowym, którego celem jest utworzenie obiektu zastępującego inny obiekt. Stosowany jest on w celu kontrolowanego tworzenia na żądanie kosztownych obiektów oraz kontroli dostępu do nich.

Behawioralne

- **Command -** traktuje żądanie wykonania czynności jako obiekt, dzięki czemu umożliwia parametryzowanie żądania w zależności od dobiorcy,
- **Interpreter -** definiuje opis gramatyki pewnego języka interpretowalnego oraz stworzenie dla niej interpretera, dzięki któremu możliwe będzie rozwiązanie problemu,
- **Iterator -** dostęp sekwencyjny do kolejnych elementów zgrupowanych w większym obiekcie lub liście obiektów,
- **Mediator -** definiuje uproszczoną komunikację pomiędzy klasami, zapewnia jednolity interfejs do różnych elementów podsystemu,
- **Memento -** służy do zapamiętania i odtwarzania na zewnątrz stanów obiektu, pozwala na przywrócenie wcześniejszych stanów obiektu,
- **Observer -** informuje zainteresowane klasy o zmianach w innych klasach,
- **Visitor -** służy do powiadamiania obiektów, które wyraziły chęć otrzymywania informacji o zmianie stanu innego obiektu,
- **State -** zmienia sposób działania obiektu, poprzez zmianę stanu, wewnętrznego tego obiektu,
- **Strategy -** tworzy wspólny interfejs z dozwolonymi operacjami oraz listę klas implementujących dany interfejs dostarczających konkretne algorytmy.
- **Chain of responsibility -** sposób przekazywania żądań do przetworzenia pomiędzy różne obiekty, w zależności od typu obiektu