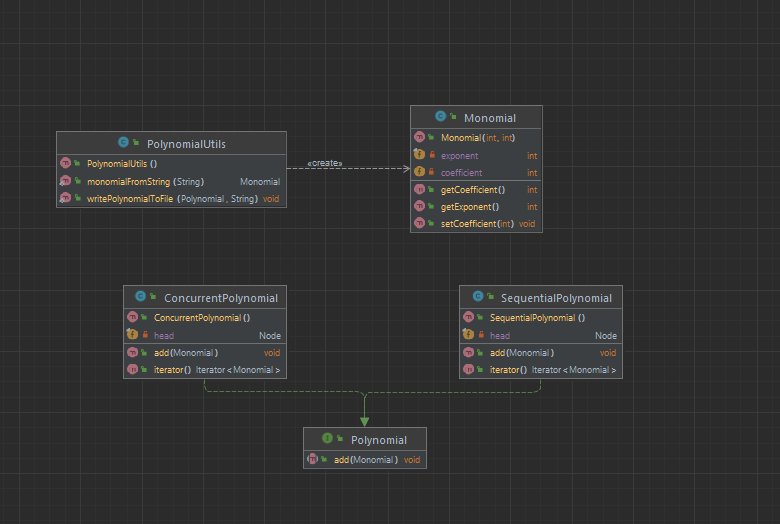
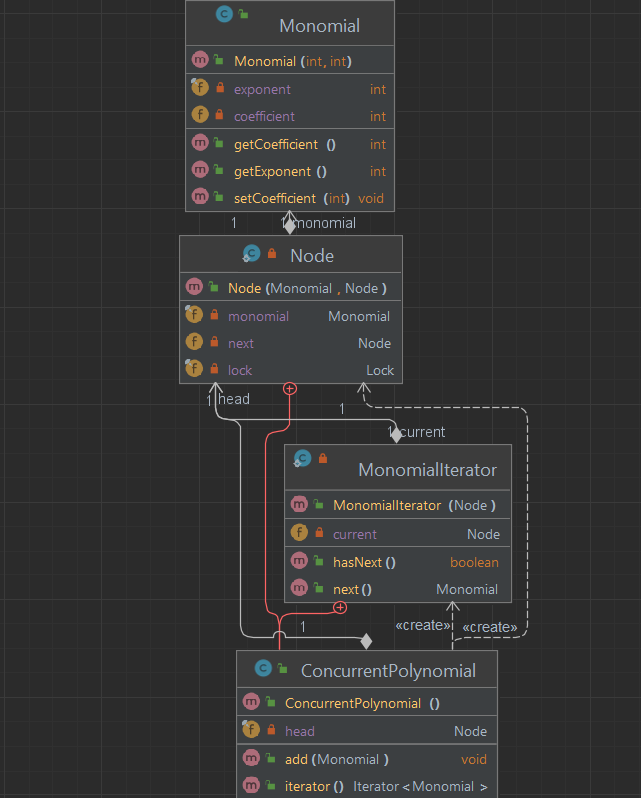
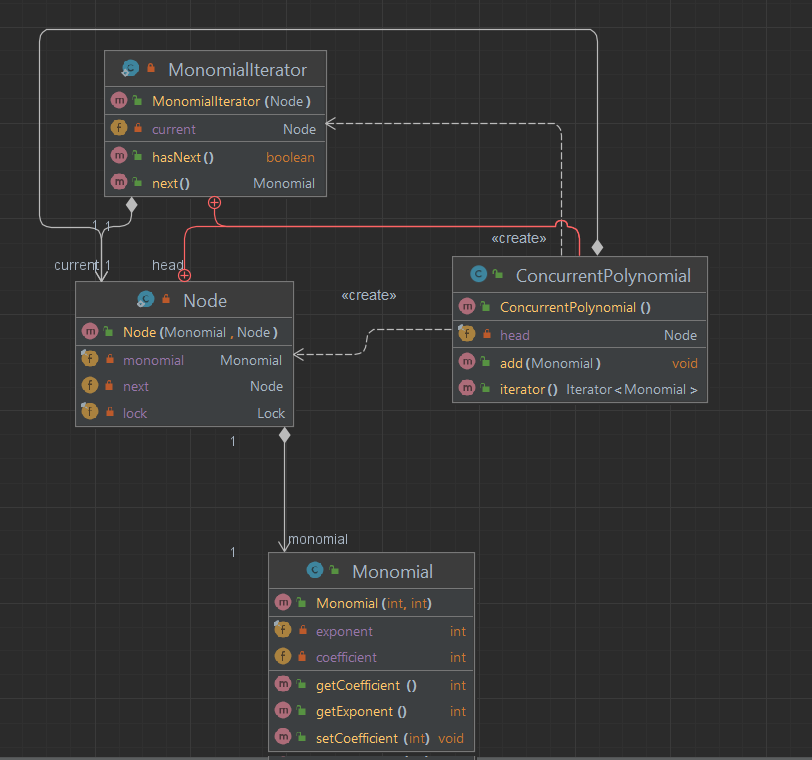
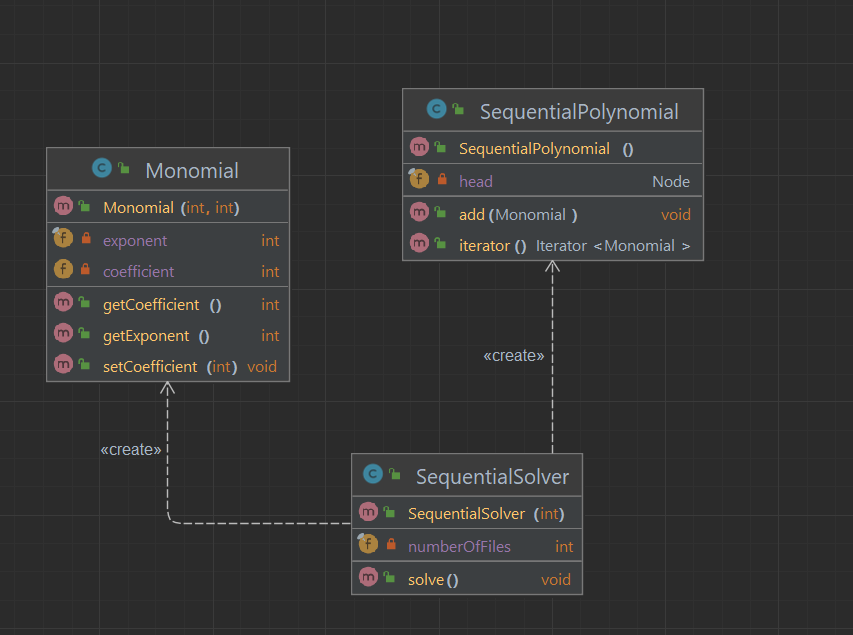
Documentatie PPD Lab 5

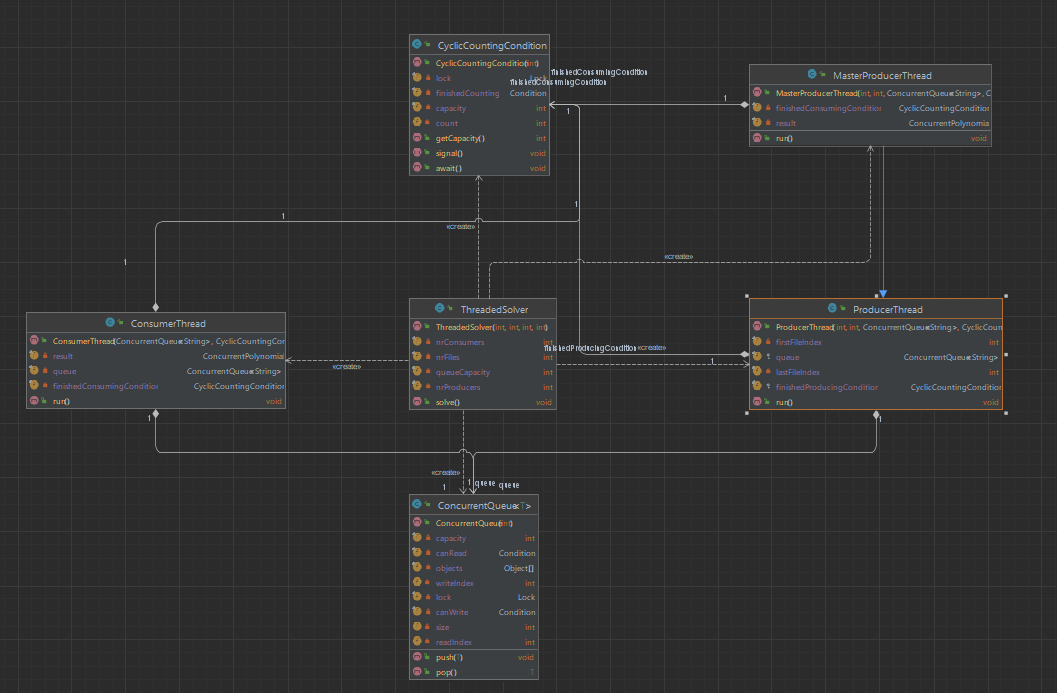
* Info:
  + OS: Windows 10
  + Processor: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz 2.30 GHz
  + Cores: 8
  + Logical Processors: 16
  + RAM: 16.0 GB (15.7 GB usable)
* Idee:
  + Fisier de intrare:
    - Fisierele de intrare au pe fiecare linie un string ce reprezinta un monom – un monom este de forma “coeficient exponent”
  + Threads:
    - Se imparte in mod egal, cu maxim o diferenta de 1, fisierele intre thread-urile producer
    - Producer threads citesc din fisiere si scriu monoamele intr-o coada de asteptare sincronizata
    - Atunci cand un producer termina de scris, semnaleaza asta
    - Consumer threads citesc din coada monoamele si le aduna la polinomul rezultat
    - Atunci cand citesc null, se opresc din citit din coada si semnaleaza ca au terminat de consumat
    - Primul producer este si master thread – dupa ce isi termina treaba de producer thread, isi incepe treaba de master
    - Master thread-ul asteapta sa fie anuntat ca toti producers au terminat de scris in coada, apoi pushuieste atatea valori de null in coada cati consumers sunt
    - Master thread-ul asteapta sa fie anuntat ca toti consumers au terminat de adunat la polinom, apoi scrie in fisierul de output polinomul rezultat
  + Coada sincronizata:
    - Producers asteapta sa poata scrie in coada – asta inseamna ca coada nu este plina
    - O data ce se poate scrie in coada, se scrie si se anunta ca se poate citi din ea
    - Consumers ateapta sa poata citi din coada – asta inseamna ca coada nu este goala sau ca nu se va mai scrie in ea
    - O data ce se poate citi din coada, se citeste si se anunta ca se poate scrie in ea
  + Polinom:
    - Un monom este format din coeficient si un exponent – fiecare monom are asociat si un Lock.
    - Asocierea se face printr-o clasa de Node
    - Polinomul are un dummy head, ce este mereu prezent si reprezinta intrarea in lista
    - In continuare, ne vom preface ca node-ul dummy are un grad de –infinit, iar null, capatul listei, are un grad de +infinit, pentru a simplifica logica
    - Avand de adunat monomul MA, vom lucra pe zone de tipul (MX -> MY) (mereu vom avea blocate maxim 2 noduri, fiind indeajuns pentru a nu realiza deadlock-uri sau probleme de logica)
    - Astfel, parcurgerea (si blocarile) vor fi: M1, M1 -> M2, M2, M2 -> M3 etc.
    - Conditia de oprire este grad(MX)<grad(MA)<=grad(MY)
    - Astfel, mereu vom avea urmatoarele posibilitati:
      * grad(MA) != grad(MY) -> vom avea MX ->MA -> MY (Valid chiar si pentru MY null)
      * grad(MA)=grad(MY) -> vom updata coeficientul lui MY. Daca acesta devine 0, il vom sterge, avand MX -> MY1 (unde MY1 era ce urma dupa MY) - este imposibil sa avem inconsistente deoarece, singurul mod in care MY1 ar fi sters ar fi daca ce era anterior lui ar fi blocat, dupa cum am descris cazurile de mai sus – asta este imposibil, deoarece am blocat deja MY
* Specificatii:
  + Specifcatiile sunt puse pe codul sursa, la fiecare metoda, impreuna cu detaliile extra de implementare
* Java:
  + Diagrame :



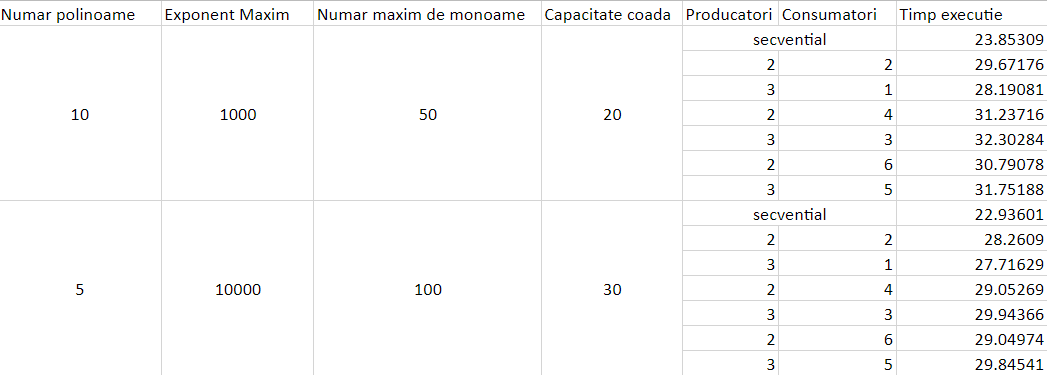








* + Rezultate:



* + Pentru Nr polinoame=10, Exponent Maxim=1000, Nr maxim monoame=50, Capacitate=20:
    - Varianta cea mai eficienta este cea sev
    - Varianta secventiala este cea mai inceata
    - Overall, cu cat avem mai multi producatori, iesim mai bine cu timpul
    - Ca si numar de thread-uri, avem o distributie normala, avand ordonarea dupa timpi: 4T < 6T > 8T
  + Pentru toate cazurile:
    - Varianta secventiala este cea mai eficienta
    - Cresterea numarului de thread-uri are impact negativ asupra performantei
    - Overall, este mai eficient sa avem un numar mai mic de producers
* Overall Conclusions:
  + Varianta paralelizata este mai slaba ca varianta secventiala
  + Comparativ cu varianta de la Lab4, timpii sunt mult mai slabi, probabil deoarece trebuie sa se tot blocheze si deblocheze cate un lock pentru fiecare nod