1. **Calitatea codului**
2. **Conformitatea cu stilul de cod Java:**
   * Verificați dacă sunt respectate convențiile de denumire Java si este pastrata consistenta notatiilor;
   * Asigurați-vă că numele variabilelor și metodelor sunt descriptive.
3. **Gestionarea excepțiilor:**
   * Asigurați-vă că sunt gestionate excepțiile adecvat în cod;
   * Justificați dacă excepțiile sunt înregistrate corespunzător sau transmise mai departe în mod adecvat.
4. **Pattern-uri:**
   * Identificati patternurile din cod;
   * Justificați dacă sunt corect folosite
5. **Verificarea metricilor OOP:**
   * Justificați respectarea metricilor definite in fisierul “metriciPOO.doc”
6. **Verificarea eficientei si performantei**
   * Justificați dacă codul are o complexitate optima
   * Justificați dacă exista alocari inutile de memorie
7. **Corectitudinea functionala a codului**
8. **Corectitudine funcțională.** 
   * Justificați dacă codul funcționează corect și rezolvă problema pentru care a fost scris;
9. **Intelegerea conceptelor**
10. **Un exemplu de {Genericitate | Polimorfism static | Polimorfism dinamic | Covarianta si contravarianta de tip }**
11. **Un exemplu de {Genericitate | Polimorfism static | Polimorfism dinamic | Covarianta si contravarianta de tip }**
12. **Un exemplu de {Genericitate | Polimorfism static | Polimorfism dinamic | Covarianta si contravarianta de tip }**

4.1 Complexitate ciclomatica

Măsoară numărul de puncte de decizie (modificare a fluxului de control) dintr-o metodă. Dacă metoda are prea multe căi, atunci devine mai dificil de testat și întreținut. Trebuie redus numarul de puncte de modificare a fluxului de control (e.g if, for, switch) din metode. SugesYe: metodele care conYn multe as[el de puncte trebuie sparte in metode mai mici.

4.2 Complexitate cognitiva

Complexitatea cognitiva masoara cat de greu este de citit si inteles un cod. Complexitatea ciclomatica adreseaza testabilitatea in Ymp de complexitatea cognitiva adreseaza intelegerea. Trebuie redus numarul de blocuri imbricate din metode (e.g un if in interiorul unui if in interiorul unui ciclu while). Trebuie redus numarul de puncte de decizie din metode. Sugestie: metodele care contin multe astfel de puncte trebuie sparte in metode mai mici.

4.3 Lungimea claselor

Prea multe linii de cod într-o clasă sugerează că se pot separa unele metode în clase diferite, făcând astfel codul mai ușor de gestionat și reutilizat. O anumită clasă face prea multe lucruri. Trebuie avut in vedere principiul „Single Responsibility” și factorizate clasele în consecință.

4.4 Lungimea metodelor

Este o indicatie că o metodă ar putea face mai mult decât sugerează numele ei. Metodele lungi sunt dificil de citit și de înțeles. Trebuie redusa lungimea metodelor prin crearea de metode ajutătoare. Nu trebuie reutilizat cod vechi. (e.g.: cod copiat și lipit) De obicei, metodele ajutătoare sunt metode care nu folosesc atributele clasei lor.

4.5 Numarul total de extinderi ale claselor

Această valoare reprezintă de câte ori a fost ex;nsa o clasa (de exemplu, de câte ori a fost folosit cuvântul "extends" într-un fragment de cod).

Trebuie gasit un sablon de proiectare sau identificate relații între clase. Există elemente comune în mai multe clase distincte? Poate ar trebui creata o clasă părinte care să conțină acele elemente.

4.6 Implementari ale interfetelor

Valoarea reprezinta de cate ori au fost implementate interfete (e.g. de cate ori a fost u;lizata construc;a „implements” in cod.). Trebuie luata in considerare o funcționalitate comună pe care o au en;tățile. De exemplu, exista o metodă numită "luptă()" în mai multe clase, dar implementată în moduri diferite? Poate ar fi u;l să fie creata o interfață "Luptător" și implementata.

4.7 Cuplare

Numar de tipuri diferite utilizate in interiorul unei clase (e.g: Caine, Pisica, Calculator)

4.8 Exista comentarii utile