4.5.1. **Principiul Echivalenţei Refolosire/Folosire (PER)** Refolosire se refera la proprietatea de livrare. Doar componentele ce pot fi uşor urmărite, pot fi refolosite eficient.Principiul exprimă faptul că un element software reutilizabil nu poate fi refolosit cu adevărat în practică dacă nu este gestionat de un anumit tip de system de distributie. În practică, nu există ”clasă reutilizabilă” fara o garantie. Va trebui integrat intregul modul.Deci fie toate clasele unui pachet sunt reutilizabile, fie nici una nu este.Martin

4.5.2. **Principiul reutilizării comune (PRC)** „Toate clasele unui pachet [library] trebuie refolosite împreună. Dacă refolosim o clasă din pachet, le refolosim pe toate.” Principiul spune că pachetele de componente reutilizabile trebuie grupate utilitate.Când depindem de un pachet, depindem de fiecare clasă din acel pachet şi în consecinţă trebuie să grupăm într-un pachet clase ce pot fi refolosite împreună; altfel vom depinde de clase ce nu au legătură cu aplicaţia noastră, şi va trebui să re-compilăm aplicaţia la fiecare nouă versiune a pachetului de care depindem (chiar dacă versiunea modifică doar acele clase de care noi nu avem nevoie).

4.5.3. **Principiul Închiderii Comune (PÎC**) „Clasele unui pachet trebuie să fie închise faţă de aceleaşi tipuri de modificări. O modificare ce afectează pachetul, afectează toate clasele acelui pachet.Principiul exprimă faptul că acele clase care se modifică împreună trebuie grupate la un loc. Modificarile trebuie să modifice un număr minim de pachete distribuite. Acest principiu este în strânsă relaţie cu Principiul Deschis- Închis întrucât ajută dezvoltatorul să definească ce parte a programului va rămâne stabile si in functie de tip. Altfel spus, clasele unui pachet trebuie să fie aplicabile şi pentru un anumit tip de modificare, fie modifica toate clasele fie niciuna.

4.5.4. **Principiul dependenţelor aciclice** „Structura de dependenţe a componentei distribuite trebuie să fie un Graf Direcţionat Aciclic . Nu trebuie să existe cicluri.”Concluzia principală a acestui Principiu este că structura de dependenţe a pachetelor trebuie creată după scrierea codului, nu înainte.

4.5.5. Principiul Dependenţelor Stabile „Dependenţele între componente trebuie să fie în direcţia stabilităţii. O componentă trebuie să depindă de componente mai stabile decât ea.”Exemplu: Programul Copy. Acolo nu există dependenţe – ceea ce conferă robusteţe, şi reutilizabilitate codului, şi, pe de altă parte, dependenţele sunt prea puţin probabil să se schimbe. Concret, clasele Reader şi Writer au volatilitate scăzută deci şi clasa Copy (care depinde de ele) este “imună” la modificări. În concluzie, o Dependenţă Bună este o dependenţă de o ţintă cu volatilitate redusă.

4.5.6. **Principiul Abstracţiunilor Stabile (PAS)** „Abstractizarea unui pachet depinde de stabilitatea sa. Pachetele stabile trebuie să fie abstracte. Pachetele instabile trebuie să fie concrete.Arhitectura Ideală ar trebui să conţină pachetele instabile (modificabile) în partea superioară iar pachetele stabile (greu de modificat) în partea inferioară. Sa fie uşor de extins şi dificil de modificat. Abstractizarea unui pachet se măsoară cu ajutorul unui număr $A \in [0, 1]$, unde valoarea 0 =absenţa claselor abstracte, val1 =at cand clasele sunt abstracte.