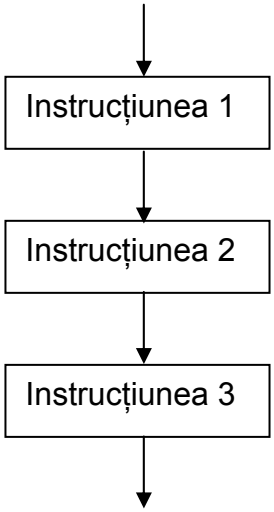
5. Condiții, expresii logice şi structuri de control pentru selecție

Până acum, instrucțiunile din programele scrise de noi se executau în ordinea în care erau scrise. Să presupunem acum că dorim să verificăm validitatea unor date de intrare şi apoi să facem un calcul sau să afişăm un mesaj de eroare, dar să nu le facem pe amândouă. Pentru aceasta va trebui să punem o întrebare şi, bazându-ne pe răspuns, să alegem una sau alta dintre variantele de evoluție a programului.

Instrucțiunea `if` ne permite să executăm instrucțiunile într-o altă ordine decât cea fizică.

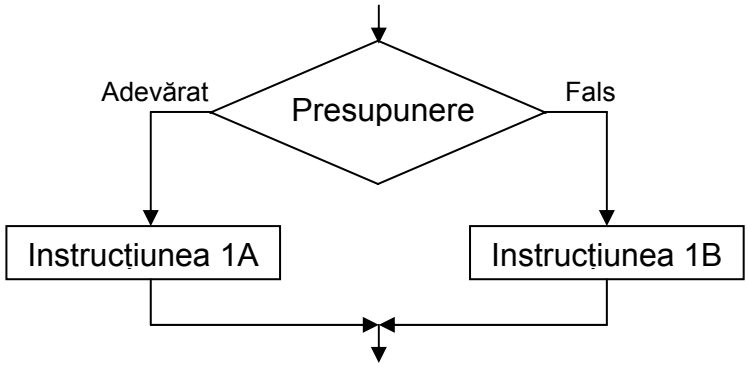
5.1 Ordinea de execuție a intrucțiunilor

La un moment dat, calculatorul se găseşte sub controlul unei instrucțiuni. După ce această instrucțiune este executată, controlul este trecut următoarei instrucțiuni. În mod obişnuit execuția este secvențială.



Acolo unde dorim ca execuția să nu mai fie secvențială, introducem *structuri de control*.

Atunci când vrem ca un program să aleagă între două acțiuni alternative, folosim o structură de control al selecției. Vom face o presupunere care poate fi adevărată sau falsă. Dacă este adevărată, calculatorul execută o intrucțiune. Dacă este falsă, execută altă instrucțiune.



De exemplu, programul poate testa la un moment dat dacă un angajat va fi plătit pentru ore suplimentare lucrate. Pentru aceasta, el va testa presupunearea că

angajatul a lucrat mai mult de 40 de ore într-o săptămână, iar dacă este falsă calculează suma obişnuită.

5.2 Condiții şi expresii logice

Pentru a pune o întrebare în C++, facem o presupunere care poate fi adevărată sau falsă. Calculatorul *evaluează* presupunearea pentru a constata dacă este adevărată sau falsă.

Expresii logice

În C++ presupunerile iau forma expresiilor logice sau booleene. O astfel de expresie este alcătuită din valori logice şi operații.

Datele booleene

Versiunile actuale ale limbajului de programare C++ includ un tip de dată numit `bool` al cărui domeniu de valori este format din constantele literale `true` şi `false`.

Exemplu
```
const bool checkValue = true;
bool dataOK;
dataOK = false;
```

Pentru compatibilitatea cu versiunile mai vechi ale standardului C++ în care acest tip de dată nu exista, valoarea `true` poate fi, de asemenea reprezentată, de orice valoare nenulă, iar `false` poate fi reprezentată de valoarea 0.

Exemplu
```
const bool checkValue = 1;
bool dataOK;
dataOK = 0;
```

Valorile booleene `false` sunt tipărite în mod implicit ca valoare `0`, iar valorile `true` sunt tipărite ca valoare `1`. Operatorul de inserție în stream << tipăreşte variabilele de tip `bool` ca întregi.

Exemplu
```
bool dataOK;
dataOK = false;
cout << dataOK << endl;                    0
```

Manipulatorul `boolalpha` setează stream-ul de ieşire ca să afişeze valorile de tip `bool` prin şirurile `true` şi `false`.

Exemplu
```
bool dataOK;
dataOK = false;
cout << boolalpha << dataOK << endl;     false
```

Operatori relaționali

Unei variabile booleene îi poate fi asignat rezultatul comparării a două valori.

Exemplu
```
bool maiMicDecat;
int i, j;
cin >> i >> j;
maiMicDecat = ( i < j );
```

Variabilei `maiMicDecat` i se asignează `true` când `i < j`.

Comparând două valori, prespunem că o relație, de exemplu *mai mic decât*, există între ele. Dacă relația există într-adevăr, presupunerea este adevărată. Dacă nu, este falsă.

În C++ putem testa următoarele relații:

Operator relațional	**Exemplu**	**Semnificație**
==	x == y	x este egal cu y
!=	x != y	x este diferit de y
>	x > y	x este mai mare decât y
<	x < y	x este mai mic decât y
>=	x >= y	x este mai mare sau egal decât y
<=	x <= y	x este mai mic sau egal decât y

În C++, rezultatul unei comparații poate fi `true` sau `false`.

Exemplu

Dacă `x` are valoarea `5` şi `y` are valoarea `10`, următoarele relații sunt adevărate:

```
x == 5
x != y
y > x
x < y
y >= x
x <= y
```

Dacă `x` este `'M'` şi `y` este `'R'`, expresiile de mai sus sunt de asemenea `true` deoarece sunt comparate codurile ASCII ale caracterelor. În acest cod, literele mari sunt aşezate inaintea celor mici.

Exemplu

`'M' < 'R'` şi `'m' < 'r'` sunt `true`

`'m' < 'R'` este `false`

Trebuie să fim atenți la tipurile de dată ale valorilor pe care le comparăm. Cel mai sigur este să comparăm int cu int, double cu double etc. Dacă nu, apar forțările implicite de tip. Ca şi în cazul expresiilor aritmetice, cel mai sigur, în aceste situații este să folosim cast explicit pentru a ne face intențiile cunoscute:

```
nrDouble >= double(nrInt)
```

sau

```
nrDouble >= static_cast<double>(nrInt)
```

De asemenea, valorile char trebuie comparate doar cu valori char.

Exemplu

`'0' < '9'` sau `0 < 9` sunt corecte

`'0' < 9` se realizează cu o forțare de tip şi rezultatul nu este cel aşteptat

Se pot folosi operatori relaționali nu doar pentru a compara valori ale variabilelor sau constantelor, ci şi pentru a compara valori ale expresiilor aritmetice.

Exemplu

Dacă avem

```
int x = 17, y = 2;
```

atunci următoarea expresie este adevărată:

```
x + 3 == y * 10
```

O eroare des întâlnită în scrierea programelor C++ este confuzia între operatorii = şi ==. Folosirea operatorului == pentru asignare sau a operatorului = pentru a testa egalitatea sunt erori logice.

Operatori logici

În matematică, operatorii logici AND, OR şi NOT acționează asupra expresiilor logice care joacă rolul de operanzi. C++ foloseşte simboluri speciale pentru fiecare dintre aceşti operatori.

Operator logic	**Exemplu**	**Semnificație**
`&&`	`x && y`	AND logic
`||`	`x || y`	OR logic
`!`	`!x`	NOT logic

Pentru ca rezultatul operației AND (`&&`) să fie `true`, ambii operanzi trebuie să fie `true`. Tabelul de adevăr pentru operatorul `&&` este:

Expresia 1	**Expresia 2**	**Expresia 1 && Expresia 2**
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Operația OR (`||`) cere ca cel puțin un operand să fie `true` pentru ca rezultatul să fie `true`. Tabelul de adevăr pentru operatorul `||` este:

Expresia 1	**Expresia 2**	**Expresia 1 || Expresia 2**
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Operatorul NOT (!) precede o singură expresie logică şi dă un rezultat opus valorii logice a expresiei. Tabelul de adevăr pentru operatorul `!` este:

Expresia 1	**! Expresia 1**
`false`	`True`
`true`	`False`

NOT ne dă o metodă de a inversa sensul unei presupuneri.

Exemplu

```
!(oreLucrate > 40)
```

este echivalent cu

```
oreLucrate <= 40
```

În unele expresii prima formă este mai clară, în altele cea de-a doua.

Exemple

Conform regulilor lui De Morgan avem:

```
!( a == b )             ⇔  a != b
!( a == b || a == c )   ⇔  a != b && a != c
!( a == b && c > d )    ⇔  a != b || c <= d
```

Evaluări condiționale

Evaluarea expresiilor logice se face de la stânga la dreapta, aceasta oprindu-se atunci când este cunoscută valoarea întregii expresii. Întrebarea care se pune este cum poate calculatorul să ştie dacă valoarea unei expresii este `true` sau `false` dacă nu a evaluat-o până la capăt.

O operație AND este `true` dacă ambii operanzi sunt `true`. Dacă în expresia

```
i == 1 && j > 2
```

`i` are valoarea `10`, subexpresia din stânga va fi `false`, deci expresia nu mai poate avea decât valoarea `false`.

Evaluarea unei expresii OR se opreşte când una dintre subexpresii este `true`, deci rezultatul este `true`.

Precedența operatorilor

Am discutat despre precedența operatorilor în evaluarea expresiilor aritmetice. Regulile de precedență se aplică şi operatorilor logici şi relaționali. Lista operatorilor şi precedența lor este urnătoarea:

Cea mai înaltă precedență

```
(   )
!
*   /   %
+   -
<   <=   >   >=
==   !=
&&
||
=
```

Cea mai scăzută precedență

Parantezele rotunde au prioritate asupra oricărei operații. Acestea trebuie folosite întotdeauna în pereche.

Operatori relaționali pentru tipuri reale

Operatorii relaționali pot fi folosiți pentru orice tipuri de dată.

Vom prezenta cazul valorilor `double`. Ca regulă generală, nu studiați egalitatea a două numere reale pentru că, datorită micilor erori care apar la calculele cu numere reale, două astfel de valori nu sunt întotdeauna egale.

Pentru instrucțiunile

```
float oTreime, x;
oTreime = 1.1 / 3.0;
x = oTreime + oTreime + oTreime;
```

Ne aşteptăm ca `x` să aibă valoarea `1.1`, dar probabil că nu va fi aşa. Prima asignare va stoca în variabila `oTreime` o aproximare lui $\frac{1}{3}$, probabil `0.366667`. Cea de-a doua asignare va stoca in variabila `x` o valoare egală cu `3*0.366667`. Dacă vom compara `x` cu `1.1` rezultatul va fi `false`. Diferența dintre valorile lui `x` şi `1.1` este un număr foarte mic, de exemplu `2.38419e-008`.

În loc să testăm egalitatea, putem considera că, dacă diferența dintre cele două numere este foarte mică, putem considera că numerele sunt egale. Putem înlocui testul de egalitate cu

```
fabs(x-1.1) < 0.00001
```

5.3 Instrucțiunea `if`

Am văzut cum se scriu expresiile logice, vom vedea acum cum putem afecta fluxul secvențial al programului. Instrucțiunea `if` permite ramificarea fluxului normal. Putem pune o întrebare şi în funcție de condițiile existente putem realiza o acțiune sau alta.

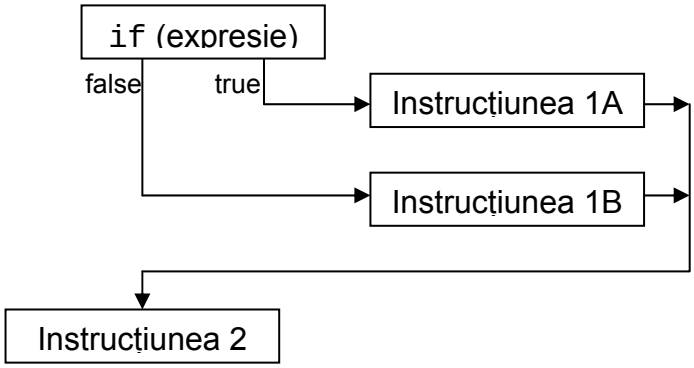
Structura IF-THEN-ELSE

În C++ instrucțiunea if poate fi folosită în două variante: forma IF-THEN-ELSE sau forma IF-THEN. O vom analiza mai întâi pe prima.

Sintaxa formei IF-THEN-ELSE este

```
if(expresie)
        Instrucțiunea 1A
else
        Instrucțiunea 1B
```

Expresia din paranteze poate avea orice tip simplu de dată. Aproape întotdeauna aceasta va fi o expresie logică. Dacă valoarea expresiei este nenulă sau `true`, calculatorul execută *Instrucțiunea 1A*. Dacă valoarea expresiei este `0` sau `false`, se execută *Instrucțiunea 1B*. *Instrucțiunea 1A* se numeşte *clauza then*, iar *Instrucțiunea 1B* se numeşte *clauza else*.



De notat că instrucțiunea `if` foloseşte doar cuvintele rezervate `if` şi `else`, chiar dacă structura se numeşte IF-THEN-ELSE.

Exemple

```
if(oreLucrate <= 40)
    plata = sumaPeOra * oreLucrate;
else
    plata = sumaPeOra * (40.0 + (oreLucrate – 40.0) * 1.5);
cout << plata;
```

În limbaj natural, această instrucțiune spune următorul lucru: *Dacă numărul de ore lucrate este mai mic sau egal decât 40, atunci calculează suma obişnuită de plată şi apoi treci la instrucțiunea de tipărire. Dacă numărul de ore lucrare este mai mare decât 40, atunci calculează suma normală şi suma suplimentară , apoi treci la instrucțiunea de tipărire.*

Blocuri

Pentru a evita împărțirea la zero într-o expresie, să presupunem că atunci când numitorul este egal cu `0` facem două lucruri: tipărim un mesaj de eroare şi încărcăm valoarea `9999` în variabila rezultat. Trebuie, aşadar, să realizăm două instrucțiuni pe aceeaşi ramură, însă sintaxa instrucțiunii if ne limitează la una singură.

Să ne amintim că orice compilator C++ tratează blocul

```
{
...
}
```

ca o singură instrucțiune. Folosind o pereche de acolade pentru a încadra secvența de instrucțiuni, instrucțiunile vor deveni un singur bloc.

Exemplu
```
if(numitor != 0)
    rezultat = numarator / numitor;
else
{
    cout << ”Impartirea la 0 nu este permisa.” << endl;
    rezultat = 9999;
}
```
Putem folosi blocuri de instrucțiuni pe ambele ramuri ale unui IF-THEN-ELSE.

Exemplu
```
if(numitor != 0)
{
    rezultat = numarator / numitor;
    cout << ”Impartirea este realizata corect.” << endl;
}
else
{
    cout << ”Impartirea la 0 nu este permisa.” << endl;
    rezultat = 9999;
}
```
După acolada care închide blocul nu se pune niciodată semnul `;`

Forma IF-THEN

Uneori dorim să realizăm o acțiune când se îndeplineşte o condiție, dar să nu se întâmple nimic atunci când condiția este falsă. Putem lăsa ramura `else` vidă.

Exemplu
```
if(a <= b)
    c = 20;
else
    ;
```
Mai simplu, putem să nu scriem deloc ramura `else`. Ajungem, astfel, la forma IF-THEN:

`if(`*Expresie*`)`
    Instrucțiune

Putem rescrie instrucțiunea din exemplul anterior astfel:
```
if(a <= b)
    c = 20;
```
Ca şi la forma IF-THEN, ramura din IF-THEN poate fi un bloc de instrucțiuni.

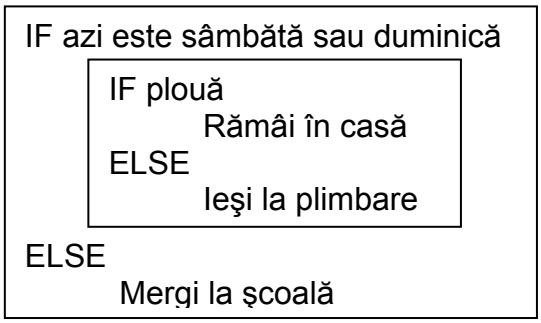
Să presupunem că avem de completat un formular pentru efectuarea unei plăți. Conform instrucțiunilor, linia 23 din formular trebuie scăzută din linia 17, iar rezultatul trebuie trecut pe linia 24. Dacă rezultatul este negativ, pe linia 24 se trece rezultatul `0` şi se bifează căsuța 24A. În C++ această cerință se implementează astfel:
```
rezultat = linia17 – linia23;
if( rezultat < 0.0 )
{
    cout << ”Bifeaza casuta 24A” << endl;
    rezultat = 0.0;
}
linia24 = rezultat;
```
Ce se întâmplă dacă uităm să folosim acoladele?

```
rezultat = linia17 – linia23;
if( rezultat < 0.0 )
    cout << ”Bifeaza casuta 24A” << endl;
    rezultat = 0.0;
linia24 = rezultat;
```

În ciuda intenţiilor noastre, compilatorul tratează clauza `then` ca una cu o singură instrucţiune. Dacă rezultatul este negativ, calculatorul execută instrucţiunea de afişare, setează rezultatul cu `0` şi apoi îl păstrează în `linia24`. Până acum totul este bine. Dacă rezultatul este pozitiv, calculatorul ignoră clauza `then` şi execută următoarea instrucţiune după instrucţiunea `if`. De aceea, rezultatul este setat cu `0`, ceea ce contravine intenţiilor noastre. Concluzia este că alinierea instrucţiunilor nu influenţează în niciun fel compilatorul.

Instrucţiuni `if` imbricate

Nu există restricţii asupra tipului de instrucţiuni pe care le poate conţine o ramură a unui `if`. Prin urmare, o ramură a unui `if` poate conţine un alt `if`. Când folosim o astfel de construcţie, spunem că am creat o *structură de `if` imbricat*.



În general, orice problemă care implică multi-ramificare poate fi codată prin instrucţiuni `if` imbricate.

Pentru a afişa numele unei zile din săptămână putem folosi o serie de instrucţiuni `if` neimbricate.

Exemplu

```
if(zi == 1)
    cout << ”Luni”;
if(zi == 2)
    cout << ”Marti”;
if(zi == 3)
    cout << ”Miercuri”;
if(zi == 4)
    cout << ”Joi”;
if(zi == 5)
    cout << ”Vineri”;
if(zi == 6)
    cout << ”Sambata”;
if(zi == 7)
    cout << ”Duminica”;
```

Acest cod poate fi rescris cu instrucţiuni `if` imbricate.

Exemplu

```
if(zi == 1)
    cout << ”Luni”;
else
```

```
if(zi == 2)
    cout << ”Marti”;
else
    if(zi == 3)
        cout << ”Miercuri”;
    else
        if(zi == 4)
           cout << ”Joi”;
        else
           if(zi == 5)
               cout << ”Vineri”;
           else
               if(zi == 6)
                   cout << ”Sambata”;
               else
                   if(zi == 7)
                       cout << ”Duminica”;
```

Este mult mai eficientă această variantă pentru că implică mai puține comparații. Structura care implementează ramificarea multiplă se numeşte IF-THEN-ELSE-IF. Putem realinia structura de mai sus pentru a evita deplasarea continuă spre dreapta datorită alinierilor convenite.

Exemplu

```
if(zi == 1)
    cout << ”Luni”;
else if(zi == 2)
    cout << ”Marti”;
else if(zi == 3)
    cout << ”Miercuri”;
else if(zi == 4)
    cout << ”Joi”;
else if(zi == 5)
    cout << ”Vineri”;
else if(zi == 6)
    cout << ”Sambata”;
else if(zi == 7)
    cout << ”Duminica”;
```

Folosirea greşită a lui else

La folosirea instrucțiunilor `if` imbricate pot apărea confuzii în legătură cu perechile `if-else`: cărui `if` îi aparține un `else`?

Să presupunem că dacă nota unui student este mai mică decât 5 dorim să tipărim ”Respins”, dacă nota lui este între 5 şi 6 să tipărim ”Admis”, iar dacă nota este mai mare decât 6 să nu tipărim nimic. Putem coda această problemă astfel:

```
if(media < 6.0)
    if(media < 5.0)
        cout << "Respins" << endl;
    else
        cout << "Admis" << endl;
```

Cum ştim cărui `if` îi aparține un `else`? O regulă din C++ spune că, în absența acoladelor, un `else` este asociat întotdeauna instrucțiunii `if` cea mai apropiată care

nu are încă un `else` drept pereche. Noi obişnuim să aliniem codul astfel încât să reflectăm această împerechere.

Ne propunem să rescriem codul de mai sus în aşa fel încât instrucțiunea `else` să fie asociată primului `if`.

```
if(media < 6.0)
    if(media < 5.0)
        cout << "Respins" << endl;
else
    cout << "Admis" << endl;
```

Această versiune nu rezolvă problema aşa cum dorim noi pentru că simpla aliniere a instrucțiunilor nu este suficientă. Conform regulii, ultimul `else` va fi împerecheat cu al doilea `if`. Pentru o soluție corectă, plasăm cel de-al doilea `if` într-un bloc de instrucțiuni.

```
if(media < 6.0)
{
    if(media < 5.0)
        cout << "Respins" << endl;
}
else
    cout << "Admis" << endl;
```

Perechea de acolade indică faptul că cea de-a doua instrucțiune `if` este completă, iar `else` trebuie să aparțină primului `if`.

5.4 Testarea stării stream-urilor de intrare / ieşire

În capitolul trecut am prezentat conceptul de stream de intrare şi de stream de ieşire în C++. Am arătat că sunt situații în care un stream poate intra in *fail state*:

- date de intrare invalide;
- tentativa de a citi un fişier dincolo de `EOF`;
- intenția de a deschide pentru citire un fişier inexistent.

C++ oferă un mecanism de a determina când un stream este in *fail state*. În expresii logice se poate folosi efectiv numele stream-ului ca şi cum ar fi o variabilă booleană.

Exemple

```
if(cin)
    ...
if(!inFile)
    ...
```

La testarea stării unui stream rezultatul poate fi nenul, adică ultima operație I/O asupra stream-ului a avut succes sau nul, când ultima operație I/O asupra stream-ului a eşuat.

5.5 Proiectarea orientată pe obiecte

Proiectarea este etapa care trebuie parcursă în procesul de dezvoltare a unui proiect cu scopul de a depăşi bariera pe care o impune complexitatea sa. O metodă de proiectare ne ajută să împărțim proiectul în module de dimensiuni mai mici, care pot fi rezolvare mai uşor.

Proiectarea orientată pe obiecte este una dintre metodele de proiectare a aplicațiilor, fiecare având propriile avantaje şi dezavantaje. În proiectarea orientată

pe obiecte problemele sunt modelate prin *obiecte*. Acestea se caracterizează prin *comportament*, adică realizează acțiuni şi prin *stări* care se modifică prin acțiuni.

Exemplu

Un *autoturism* poate fi tratat ca un obiect. Acesta are o *stare* în care prespunem că *motorul este pornit* şi un *comportament* - *pornirea autoturismului* - care îi *schimbă starea* din *motor oprit* în *motor pornit*.

Abstractizarea

În proiectarea orientată pe obiecte, complexitatea problemelor este tratată prin abstractizare. Abstractizarea înseamnă eliminarea elementelor nerelevante şi amplificarea elementelor esențiale.

Un exemplu de abstractizare este descrierea funcționării unui autoturism.

Dorim să învățăm pe cineva să conducă autoturismul folosindu-ne de abstractizare. Amplificăm elementele esențiale învățându-l pe elevul nostru cum se porneşte maşina şi cum se mânuieşte volanul. Eliminăm detaliile legate de funcționarea motorului şi nu îi vom spune că, pentru ca maşina să funcționeze, combustibilul este pompat din rezervor către motor unde printr-o scânteie acesta se aprinde, are loc o mică explozie care împinge un piston care, la rândul său, face ca motorul să se mişte.

Problemele au, în general, mai multe nivele de abstractizare. Putem discuta despre un autoturism din punctul de vedere al şoferului, plasându-ne la un nivel înalt de abstractizare. Atunci când vom discuta cu un mecanic auto, acesta va avea nevoie de informații detaliate despre funcționarea motorului sau a instalației electrice şi atunci ne vom afla la un nivel mai scăzut de abstractizare, devenind puțin mai concreți.

Se poate vorbi, totuşi, despre abstractizare şi la nivelul mecanicului auto. Mecanicul testează sau încarcă bateria automobilului fără să se uite în interiorul acesteia, acolo unde au loc reacții chimice complexe. Pe de altă parte, un proiectant al bateriei este interesat de aceste detalii, dar nu şi de modul în care curentul generat de baterie face ca aparatul de radio să funcționeze.

Observăm astfel că putem analiza detalii cum ar fi proiectarea bateriei sau a aparatului de radio. Analiza se face pe module care pot fi mai uşor de descris, dar prin abstractizare şi ignorarea detaliilor putem să privim întregul autoturism ca un modul.