Algoritamska dekompozicija

```
enum OpCode { add, sub, …};

int main () {

  int out;
  string cmdin;
  OpCode cmdout;

  while (readCommand(out,cmdin)) {
    translate(cmdin,cmdout);
    performCmd(out,cmdout);
  }

}
```

Prenos argumenta *po referenci* (*by reference*)

```
bool readCommand (int& out, string& cmd) {

  bool ret = readOut(out);
  if (!ret) return false;

  return readCmd(cmd);

}

bool readOut (int& out) {

  out = 0;

  char c = getChar();
  while (isDigit(c)) {
      out = out*10 + (c-'0');
      c = getChar();
  }

  return (c!=EOF);

}

bool readCmd (string& cmd) {
  …
}
```

```
struct Stack {
  int stack[MaxStackSize];
  unsigned sp;
};

Stack sm[N];
…

void translate (string cmdin, OpCode& cmdout) {
  if (cmdin=="ADD") cmdout = add;
  if (cmdin=="SUB") cmdout = sub;
  …
}

void performCmd (int out, OpCode cmd) {
  switch (cmd) {

    case add: {
      int op1 = pop(out);
      int op2 = pop(out);
      push(out,op1+op2);
      break;
    }

    case sub: …
    …
  }
}

int pop (int out) {
  if (sm[out].sp==0) return 0;
  else return sm[out].stack[--sm[out].sp];
}

…
```

Algoritamska dekompozicija

- U OOP algoritamska dekompozicija ima podjednako važnu ulogu, mada ne više centralnu i nije jedini element, već sastavni element *objektne dekompozicije*
- Kao i u proceduralnom programiranju, algoritamska dekompozicija se koristi za razlaganje određenog postupka, ali uz jednu bitnu razliku: u okviru razlaganja postupka, za svaki korak definiše se *činilac* koji je *odgovoran* za izvršavanje tog koraka
- Koraci se raspodeljuju činiocima (apstrakcijama, klasama) na osnovu njihovih ranije definisanih opsega odgovornosti, ali je moguće i obratno: dekompozicijom na korake se raspodeljuju odgovornosti po apstrakcijama, ali se mogu i uvoditi ili identifikovati nove apstrakcije
- Zbog toga se postupci dekomponovani na korake i raspoređeni kao odgovornosti po klasama nazivaju češće *scenarijima* (*scenario*) ili *mehanizmima* (*mechanism*)