Hijerarhijska dekompozicija

- Posmatrajmo sledeći zahtev:
  - realizujemo apstraktnu strukturu podataka *lista* (*list*), koja će imati operacije smeštanja novog elementa na proizvoljnu poziciju u listi, iza nekog drugog elementa u toj listi, i uzimanja elementa sa proizvoljnog mesta u listi
  - želimo da obe operacije budu kompleksnosti $O(1)$, pa ćemo koristiti dvostruko ulančanu listu
  - ne želimo da za strukture pokazivača koje koristimo dinamički alociramo potreban prostor, već želimo da te pokazivače ugradimo u same objekte koji će biti ulančavani, koji god da su; zato nećemo koristiti šablone, pa ni one bibliotečne
  - pravimo *spisak zadataka* (*task list*), u koju ćemo smeštati *zadatke* (*task*)
  - *zadatak* ima i mnoge druge osobine, ponašanje, koristi se u mnogim drugim, različitim kontekstima aplikacije itd.

```
class ListElem {
public:
  void insert (ListElem* prev, ListElem* next);

protected:
  ListElem () { prev = next = nullptr; };

private:
  friend class List;
  ListElem *prev, *next;
};

void ListElem::insert (ListElem* p, ListElem* n) {
  if (p) p->next = this;
  if (n) n->prev = this;
  this->prev = p;
  this->next = n;
}

class List {
public:
  List () { head = tail = nullptr; }

  void addAtHead (ListElem* e);
  void addAtTail (ListElem* e);
  void addAfter  (ListElem* e, ListElem* prev);

private:
  ListElem *head, *tail;
};

void List::addAfter (ListElem* e, ListElem* p) {
  if (!e) return;
  if (!p) insertAtHead(e);
  else
    if (!p->next) insertAtTail(e);
    else e->insert(p,p->next);
}
```

Hijerarhijska dekompozicija

- Da bi *zadatak* (*Task*) bio umetan u listu, mora da "*umeša*" (*mix in*) strukturu i ponašanje klase *ListElem*:

```
class Task : public ListElem {…};
```

- Dakle, klasa *ListElem* jeste apstraktna klasa, koja ima i strukturu i definisane (barem neke) metode i koja predstavlja potreban *interfejs* koji neka druga klasa treba da zadovolji, da bi učestvovala u nekom mehanizmu (ovde, da bi bila umetana u listu)
- Ovakve klase nazivaju se *mixin* klase
- Pored ovog interfejsa, klasa *Task* može imati druge takve interfejse za potrebe drugih konteksta (mehanizama, scenarija) u kojima učestvuje, pa će biti *izvedena* iz više takvih klasa:

```
class Task : public ListElem, public Runnable, public Drawable {…};
```

- U potpuno agresivnom i doslednom sprovođenju principa apstrakcije, mogu se praviti ovakvi interfejsi za *svaki* pojedinačan kontekst u kome klasa učestvuje, odnosno za svaku vrstu klijenta te klase - svaki takav interfejs biće predstavljen posebnom apstraktnom klasom
- Ovakav pristup naziva se *princip segregacije interfejsa* (*interface segregation*)
- Klasa koja *zadovoljava* (implementira, *implement*) sve te interfejse je onda izvedena iz svih tih klasa
- U suprotnom, ako se ne bi radilo tako, sva svojstva i ponašanje potrebni za različite kontekste bili bi ugrađeni neposrednu u tu klasu i pomešani u njoj, pa bi ona mogla da postane glomazna i teža za razumevanje i održavanje