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

Hijerarhijska dekompozicija

- Ovakav pristup dosta se koristi i u složenim programima na jeziku C (npr. implementaciji operativnih sistema) na sledeći način:

```
struct ListElem {...};
struct Runnable {...};
struct Drawable {...};

struct Task {
  ListElem listElem;
  Runnable runnable;
  Drawable drawable;
  ...
};
```

- Dakle, instanca strukture *Task* u sebi sadrži podstrukture koje predstavljaju odgovarajuće interfejse. Kada instancu strukture *Task* treba koristiti u nekom od ovih konteksta, dostavlja se pokazivač na odgovarajuću ugrađenu podstrukturu:

```
Task* aTask = ...;

List* taskList = ...;

addAtTail(taskList, &aTask->listElem);
```

- Sa idejom da podrži ovakve načine korišćenja, ali i da implementacija u njima bude podjednaka (i podjednako efikasna) kao ova na jeziku C, jezik C++ zapravo ima koncept *izvedenih* klasa (*derived class*), sa sledećim značenjem:
  - klasa može biti *izvedena* iz više osnovnih klasa koje su navedene u zaglavlju definicije klase, iza dvotačke
  - svaki objekat izvedene klase u sebi sadrži po jedan podobjekat svake od tih osnovnih klasa
  - specifikator pristupa (*public*, *protected*, *private*) označava dostupnost dog podobjekta, na isti način kao i za članove te klase