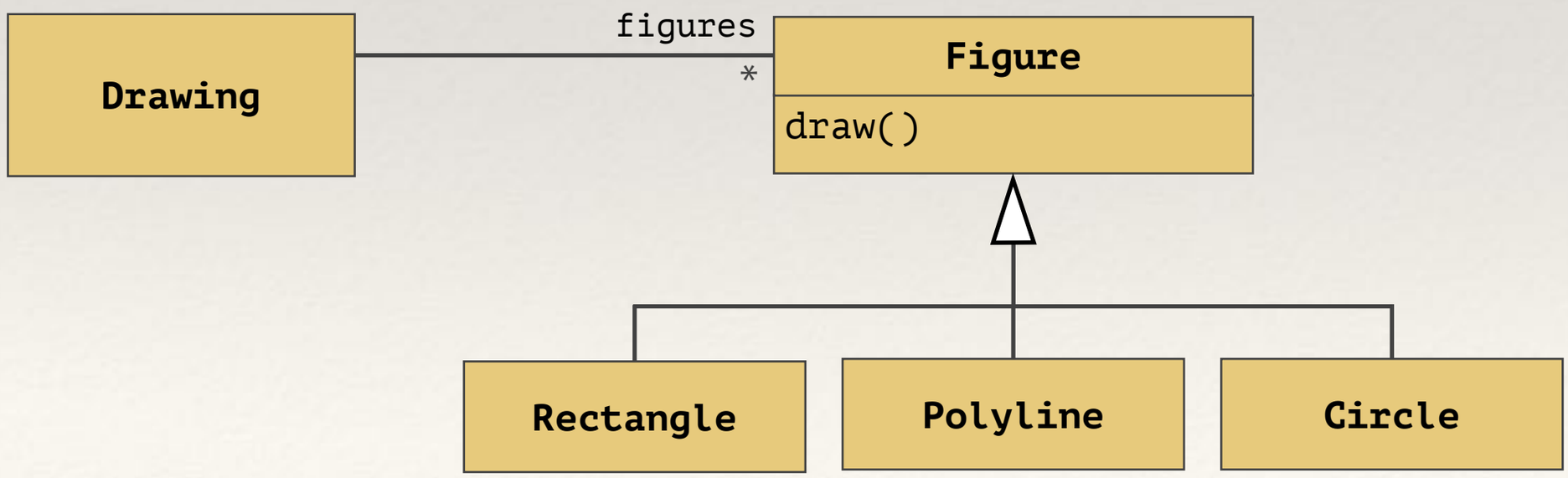
Hijerarhijska dekompozicija

- U nekom proceduralnom jeziku bismo se oslonili na već pokazano, klasično rešenje:

```
switch (fig->kind) {
  case rectangle: drawRectangle(fig,vp);
  case polyline: drawPolyline(fig,vp);
  case circle: drawCircle(fig,vp);
  …
}
```

- Pošto je *Crtež* (*Drawing*) klijent koji različite figure posmatra na isti način, jer sve želi da ih *nacrta* (*draw*), potrebna nam je generalizacija različitih vrsta figura
- Ova generalizacija onda grupiše i zajedničke osobine i zajednički interfejs posebnih klasa:
  - činjenicu da se mogu smeštati na crtež
  - uslugu da se iscrtaju



Hijerarhijska dekompozicija

- Jedna od osnovnih tendencija u OO programiranju jeste upravo generalizacija: apstrahovanje zajedničkih svojstava i formiranje uopštenih interfejsa za klijente, kako bi oni bili što manje zavisni od specifičnosti pojedinačnih slučajeva
- Specifičnosti se onda sakrivaju iza polimorfnih operacija i ugrađuju u njihove različite implementacije (metode) u izvedenim klasama
- Na ovaj način se sprege između delova softvera čine labavijim, jednostavnijim za kontrolu, a klijenti čine nezavisnijim i time fleksibilnijim (“*the less you know, the better*”)
- Ovo se ponekad naziva *principom inverzije zavisnosti* (*dependency inversion principle*):
  - u proceduralnom programiranju, u algoritamskoj dekompoziciji, potprogram (apstrakcija) na višem nivou i krupnije granularnosti zavisi od potprograma (apstrakcije) na nižem nivou, jer je poziva kao deo svoje implementacije
  - u OOP, tendencija je da jedna apstrakcija, klijent (*Drawing*) zavisi od druge uopštene apstrakcije na višem nivou (*Figure*), a da specijalizacije (*Rectangle*, *Circle*, …), kao pojedinačni slučajevi, zavise od opštije apstrakcije (*Figure*) - obrnuta zavisnost
- Generalizacijom i korišćenjem polimorfizma se drastično smanjuje količina uslovnih grananja na osnovu tipa objekta kojim se rukuje