TNM094 – Medietekniskt kandidatprojekt

Design-mönster

Design-mönster

- Problem – Kontext – Lösning
  - generella, domän-agnostiska problem och lösningar
  - kända, praktiska, testade, bra lösningar
  - kända problem-specifika problem och fallgropar
- Hjälper till i design-processen
  - för att hitta en bra, passande design
  - för att undvika vanliga misstag
  - för att undvika dålig, suboptimal implementation
- Förenklar kommunikation
  - i team-diskussioner
  - i dokumentation

[Gamma et al]

Mönster dyker upp på olika nivå

- Arkitekturella stilar
  - fundamentala strukturer utan funktioner och klasser
  - beskriver subsystem, deras ansvar och relationer
- Design-mönster (Design Pattern)
  - beskriver hur komponenter ska struktureras
  - på moduldesign-nivå
- Idiom
  - mönster på låg nivå (programrader)
  - språk-specifika lösningar

Ett design-mönster

- Namn
  - Ett unik namn som associerar till lösningen
- Problem and kontext
  - Beskrivning av problemet som ska lösas
  - Kontext och andra omständigheter som förutsätts för designen
- Abstrakt lösning
  - Modeller och beskrivningar av en programdesign
- Konsekvenser
  - Kända egenskaper och konsekvenser
- Implementationsdetaljer
  - Särskilda överväganden, variationer, fallgropar, etc.

[Gamma et al]

Mönster löser problem

- Användande av design-mönster startar med problem
  - T ex när man skriver om en bit kod
  - Använd inte design-mönster utan ett problem att lösa
- Kan problemet lösas av ett design-mönster?
  - Kan vi abstrahera problemet och definiera det tydligt?
  - Handlar problemet om hur vi strukturerar programmet?
  - eller är vi istället ute efter en ”algoritm”?
- Till exempel

  ”vi behöver
  - tillåta olika kombinationer av egenskaper hos objekt
    men
  - undvika en klass för varje kombination”

    → Decorator Pattern

Mönster ska passas in

- Förutsättningar
  - själva problemet – hur löses det
  - i vilken kontext – system-specifika detaljer
- Konsekvenser
  - hur påverkar ett mönster andra krav och egenskaper?
  - hur påverkas kvalitetskrav?
  - kan vi ändra oss i framtiden, utan för mycket arbete?
- Skapa en egen program-design utifrån mönstret
  - konkretisera i din egen kontext

Klassificering av design-mönster

- Creational – abstraherar instantiering
- Structural – bygger sammansatta strukturer
- Behavioural – algoritmer och roller
- Synchronization – säkert delade data
- Concurrency – effektiv multi-trådning
- (etc...)

[Gamma et al, Schmidt et al]

Patterns de Jour

- Mix av intressanta och användbara mönster
  - Vanligt förekommande mönster
  - Passande inom medieteknik
    - Multi-media-system, GUI, VR, visualisering, etc

- Structural
  - Facade
  - Composite
  - Decorator
- Synchronization
  - Scoped Lock
  - Thread-safe Interface

- Concurrency
  - Active Object
- Behavioural
  - Observer

[Gamma et al, Schmidt et al]

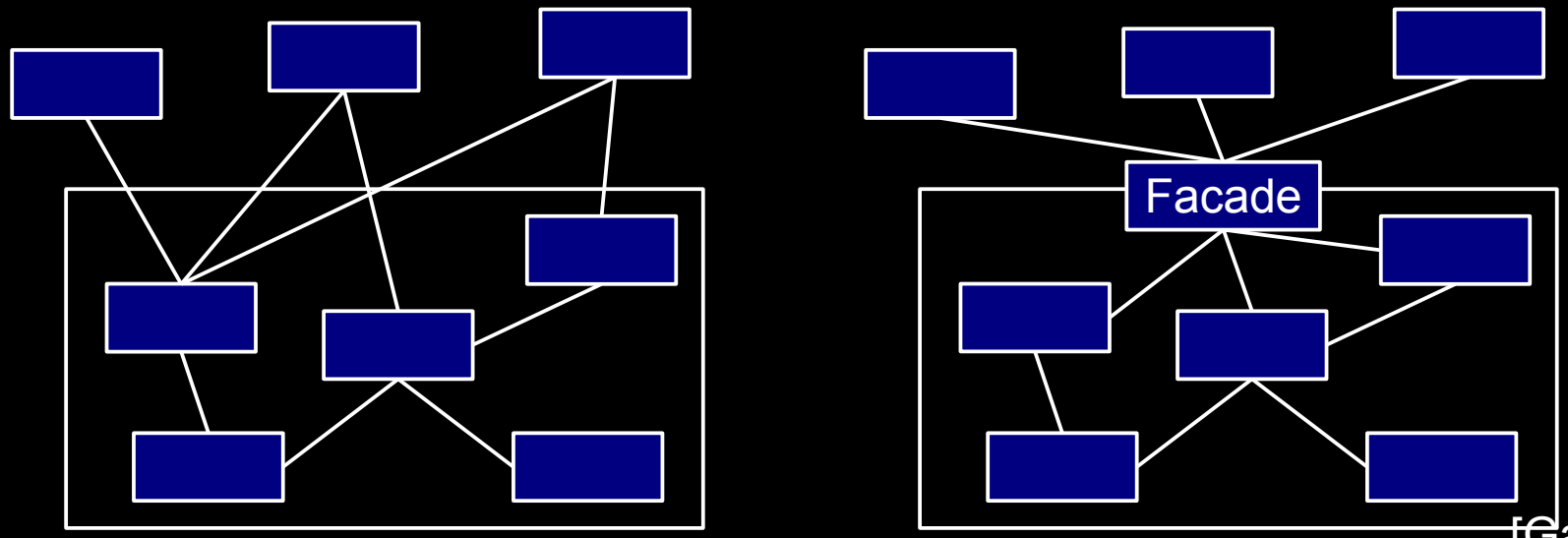
Facade

- Tillämpning
  - vill ha ett enkelt gränssnitt till en komplicerad enhet
  - vill göra enheter oberoende av en annan enhet
  - vill bygga en lager-arkitektur
- Fördelar
  - leder till svag koppling
  - gör subsystem lättare att använda
  - förhindrar inte direkta anrop

[Gamma et al]

Facade

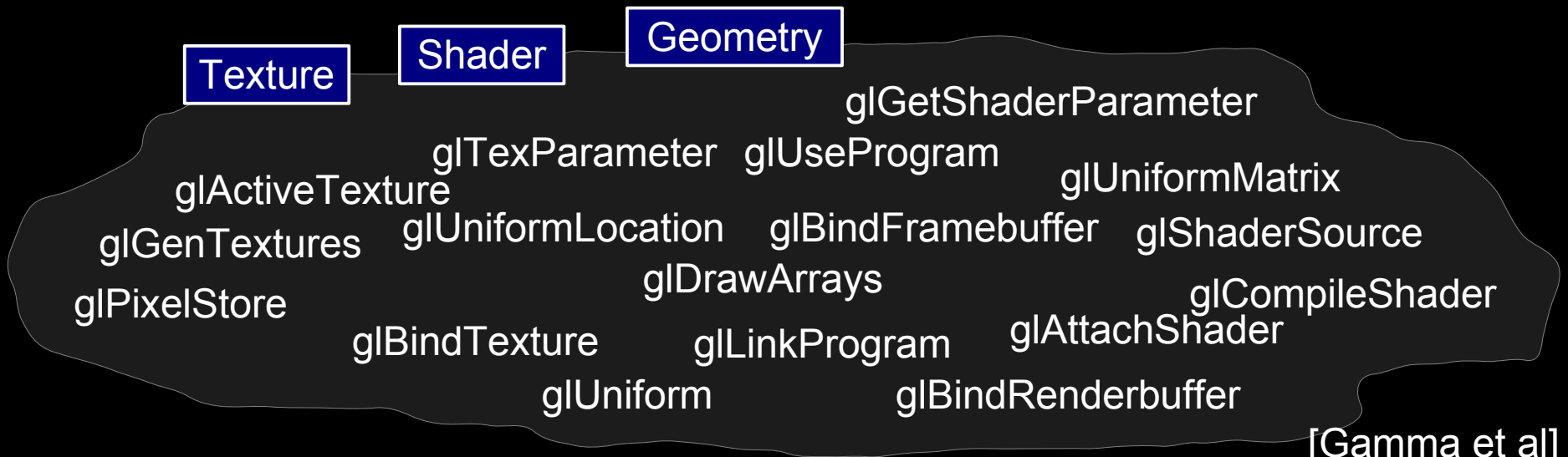
- Parts
  - Facade
    - knows subsystem classes and responsibilities
    - encapsulate objects, states and handles
    - translates and forwards calls
  - Subsystem classes
    - do the actual work



[Gamma et al]

Facade-exempel

- Kapsla in OpenGL
  - OpenGL är flexibelt och kraftfullt men komplicerat
  - OpenGL är designat för kontroll, inte för systemutveckling
  - Låt en Facade hantera tillstånd och kommunikation



[Gamma et al]

Facade

- Liknar Adapter / Wrapper
  - Förändrar/justerar en annan komponents gränssnitt
  - Anrop till en Adapter konverteras och vidarebefodras

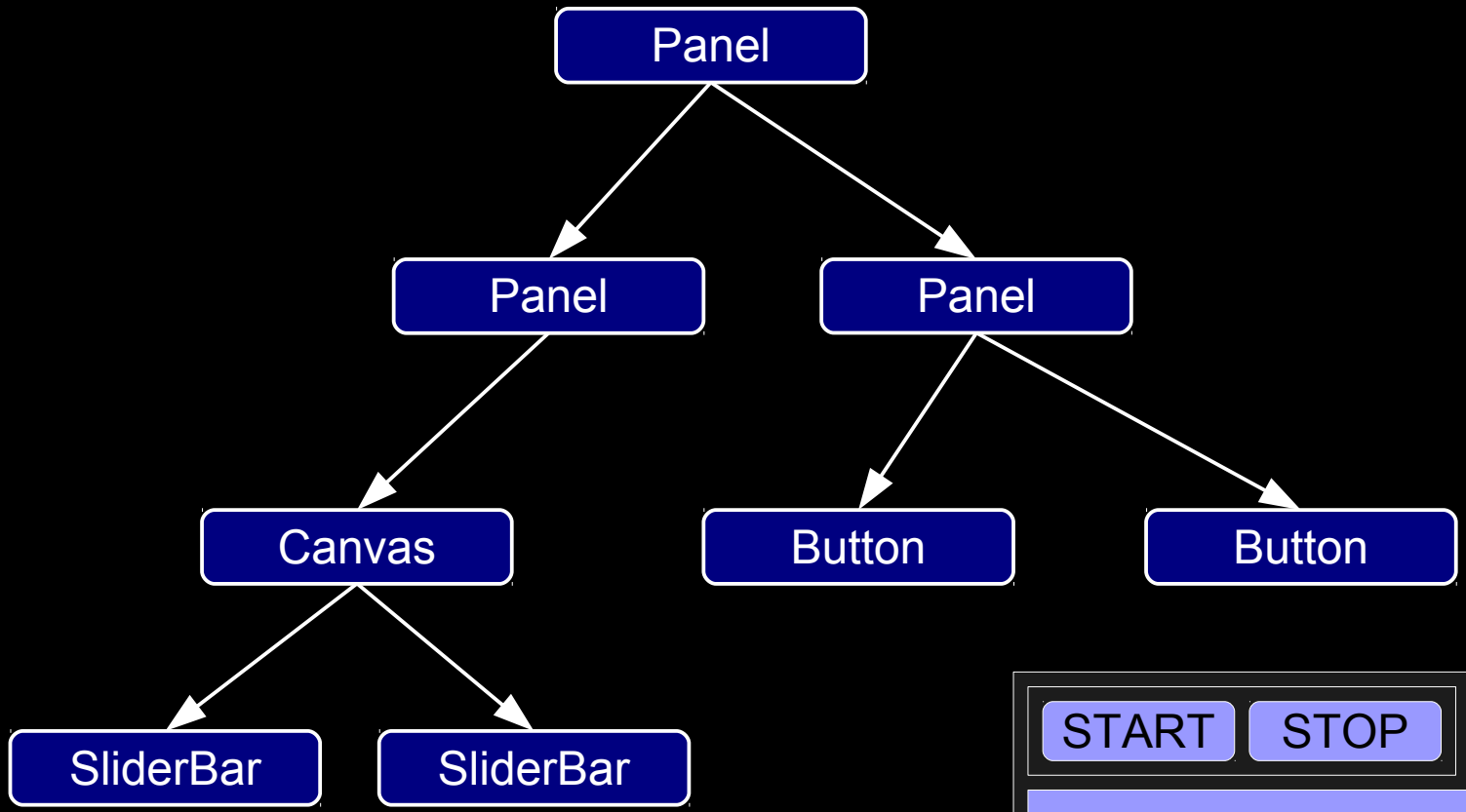
[Gamma et al]

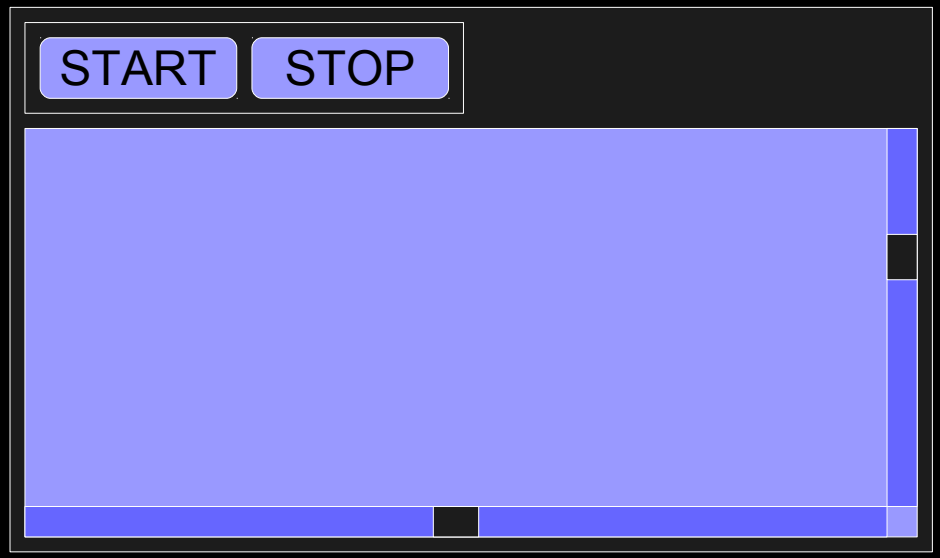
Composite

- Tillämpning
  - vill behandla sammansatta och individuella objekt likvärdigt
  - vill kunna sätta samman hierarkier av objekt
- Fördelar
  - primitiva objekt kan kombineras till komplexa rekursivt
  - kan lätt byggas ut med nya komponenter
- Nackdelar
  - kan resultera i en över-generell design
  - kräver kör-tids-kontroll för att begränsa vissa kombinationer

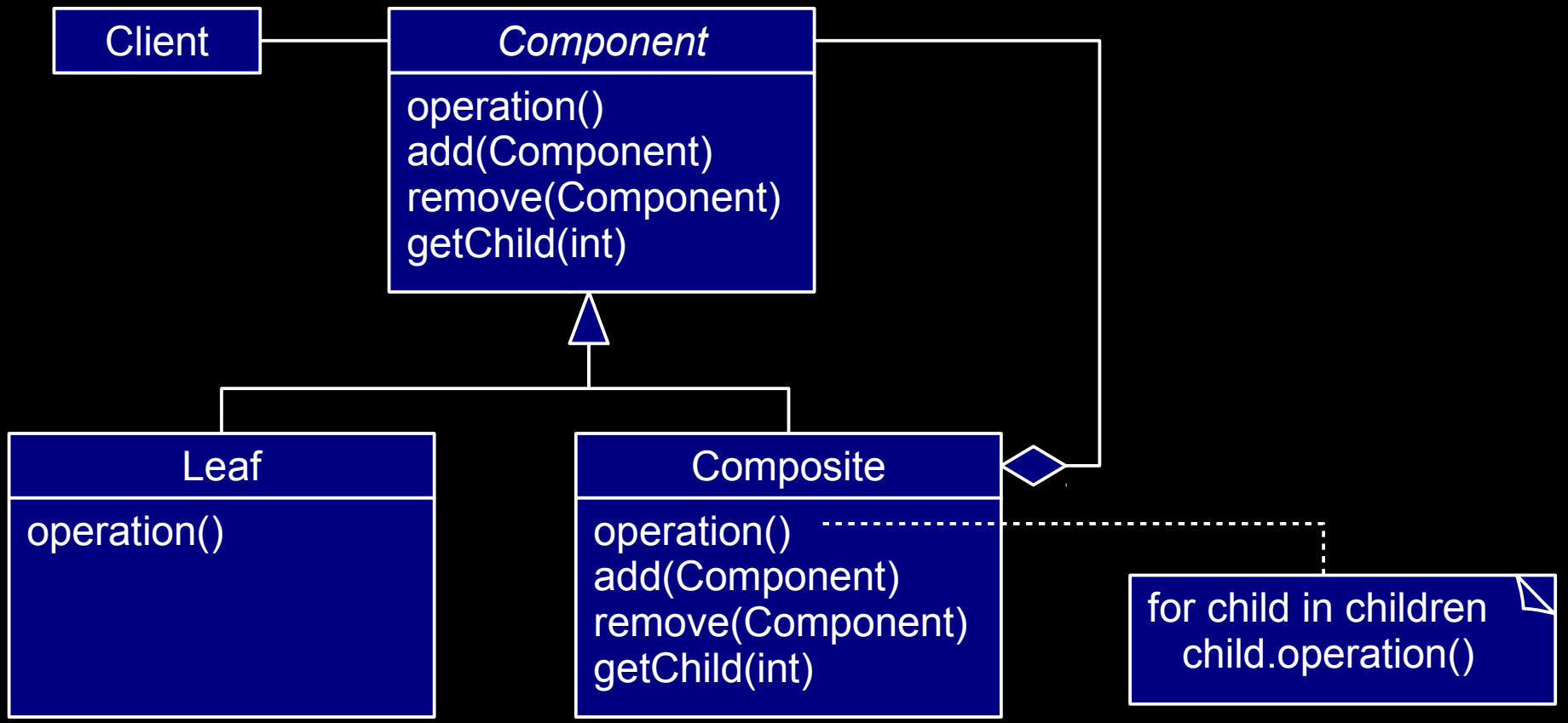
[Gamma et al]

Composite-exempel





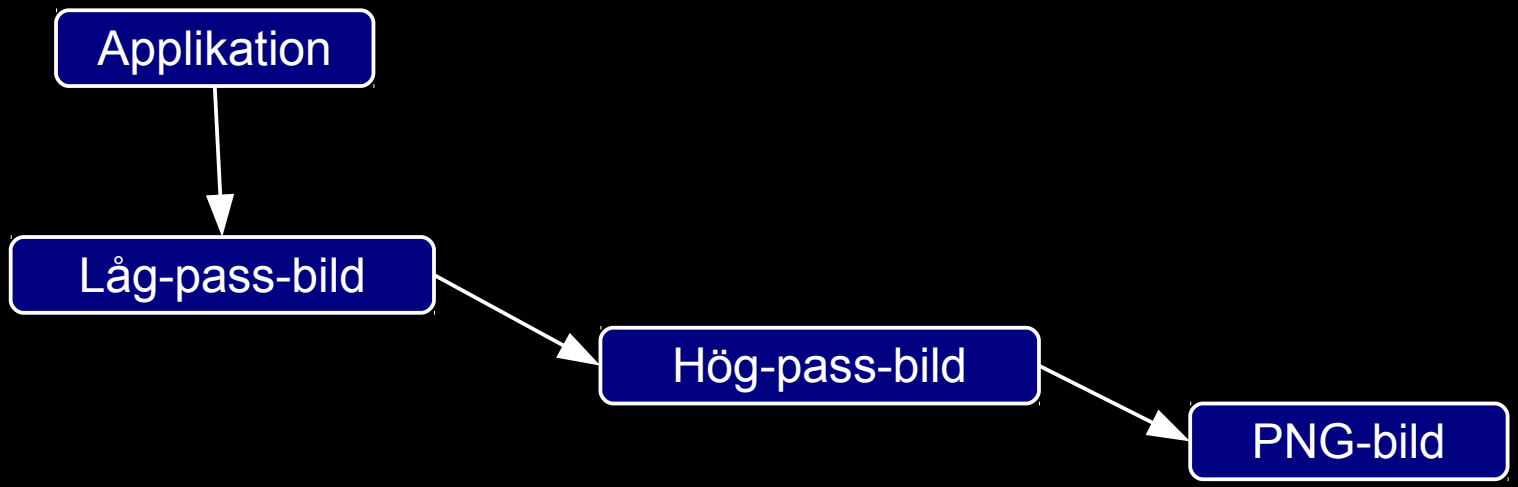
Composite


Client
Component
operation()
add(Component)
remove(Component)
getChild(int)
Leaf
operation()
Composite
operation()
add(Component)
remove(Component)
getChild(int)
for child in children
child.operation()

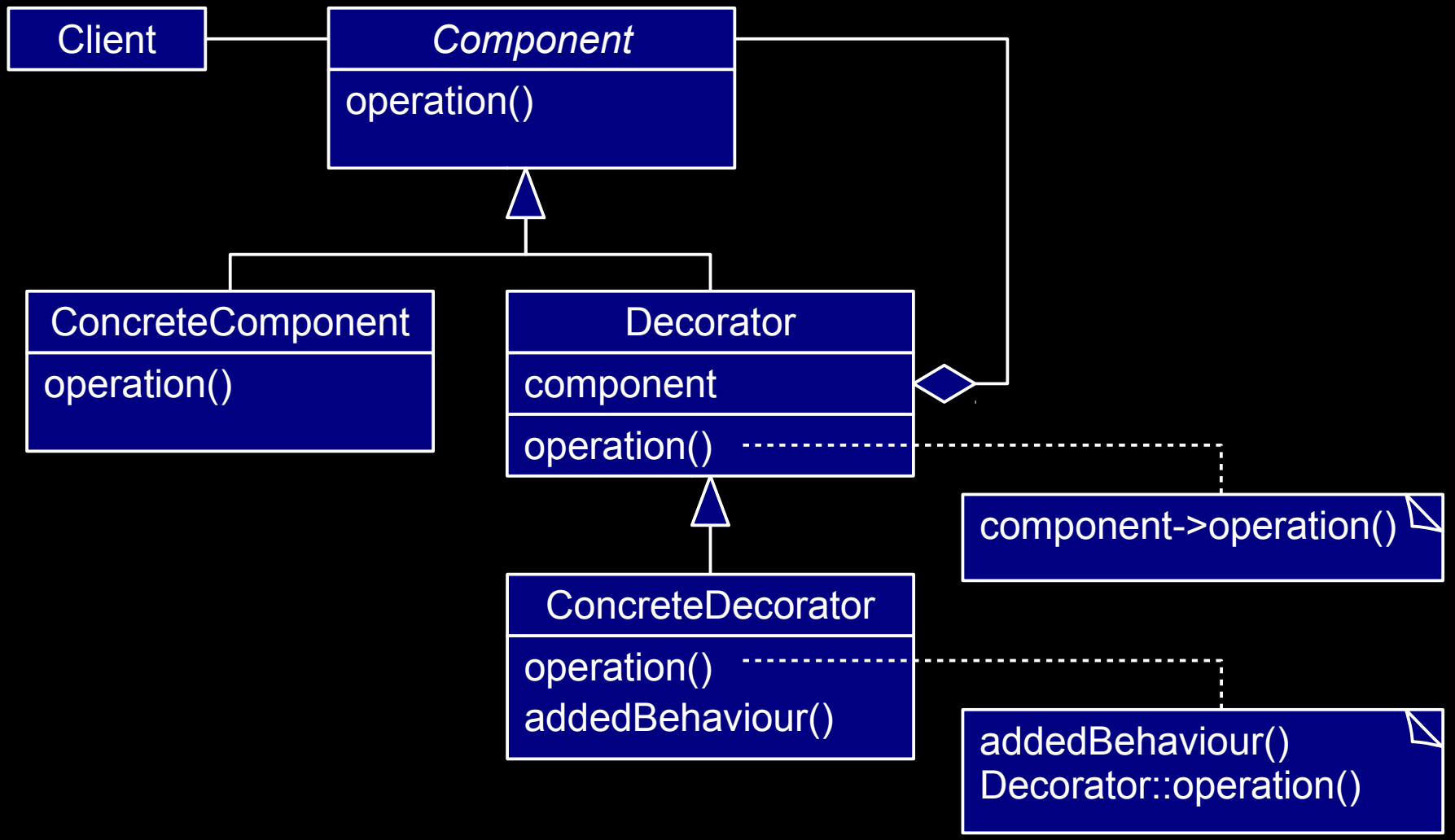
Decorator (Wrapper)

- Tillämpning
  - will dynamiskt och transparent utvidga objekts egenskaper
  - vill undvika sub-klasser för alla kombinationer av egenskaper
- Fördelar
  - mer flexibel än statiska arv
  - undviker stora, avancerade bas-klasser
- Nackdelar
  - beteendet förändras men inte objektets identitet
  - många små klasser/objekt – svårt att förstå och felsöka

Decorator (Wrapper)



Decorator (Wrapper)



Scoped Lock (Guard)

- Tillämpning
  - när man tillfälligt använder en resurs som behöver låsas
  - och det finns flera vägar ur nuvarande block
- Fördelar
  - förbättrad robusthet med enklare programkod
- Nackdelar
  - (finns men är bara vid extremt avancerade fall)

Scoped Lock (Guard)

```
class MutexGuard {
public:

  MutexGuard(pthread_mutex_t *lock)
    : _lock(lock){
    pthread_mutex_lock(_lock);
  }

  ~MutexGuard(){
    pthread_mutex_unlock(_lock);
  }

protected:
  pthread_mutex_t *_lock;
};
```

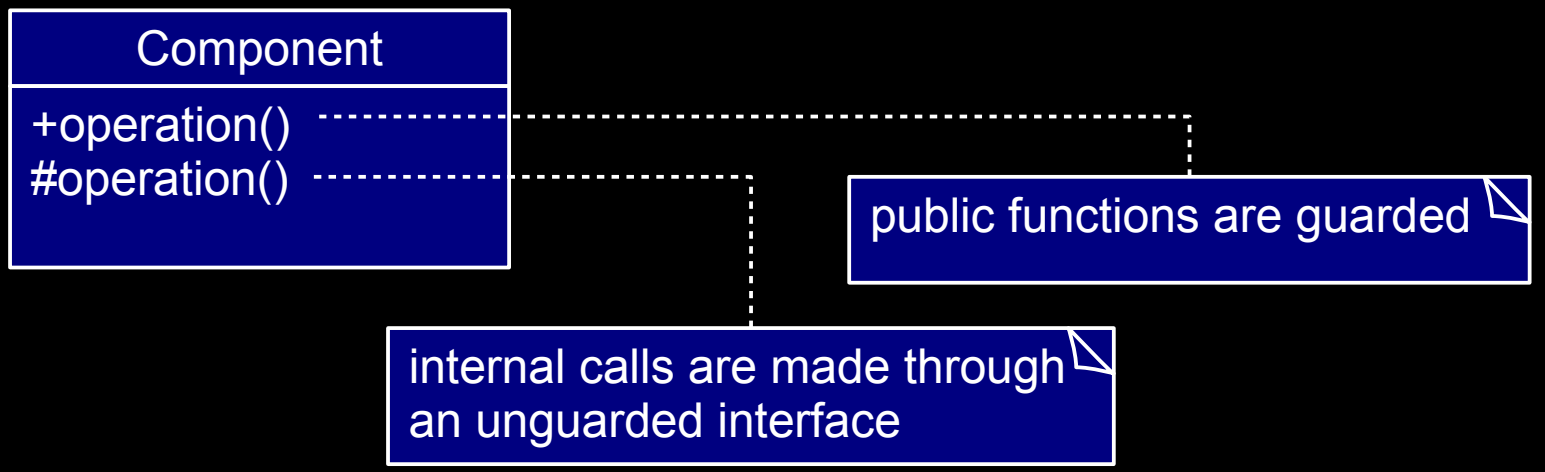
```
bool do_processing(){

  // Do some processing

  if( have_new_data ){

    MutexGuard data_guard(data_lock);

    if( !have_integrity(data) ){
      return false;
    }

    recalculate(data);
  }

  // Do other stuff

  return true;
}
```

Thread-safe Interface

- Tillämpning
  - när en enhet får anrop från olika trådar
  - och det finns flera interna anrop
- Fördelar
  - förbättrad robusthet, enklare kod och bättre prestanda
- Nackdelar
  - indirekta anrop och många extra metoder
  - dålig granularitet i synkroniseringen – sämre prestanda

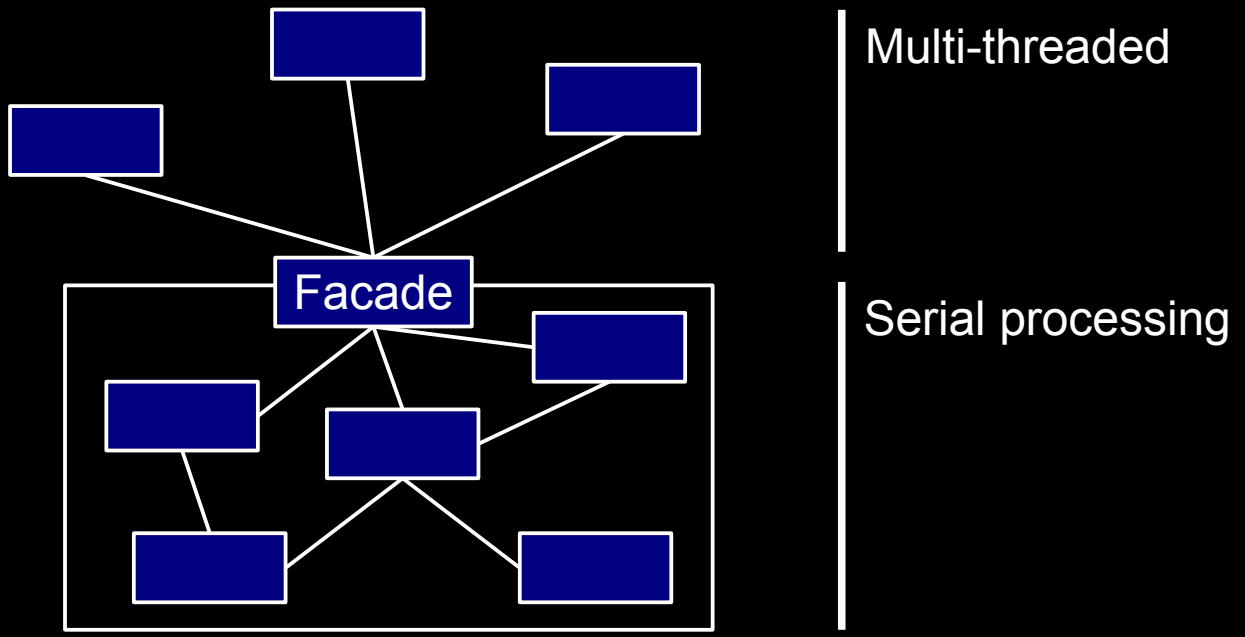
Thread-safe Interface

1. Separera interna och externa metoder
2. Skapa en dubbel uppsättning vid behov
3. Sätt ”guard” på externa metoder



Variant

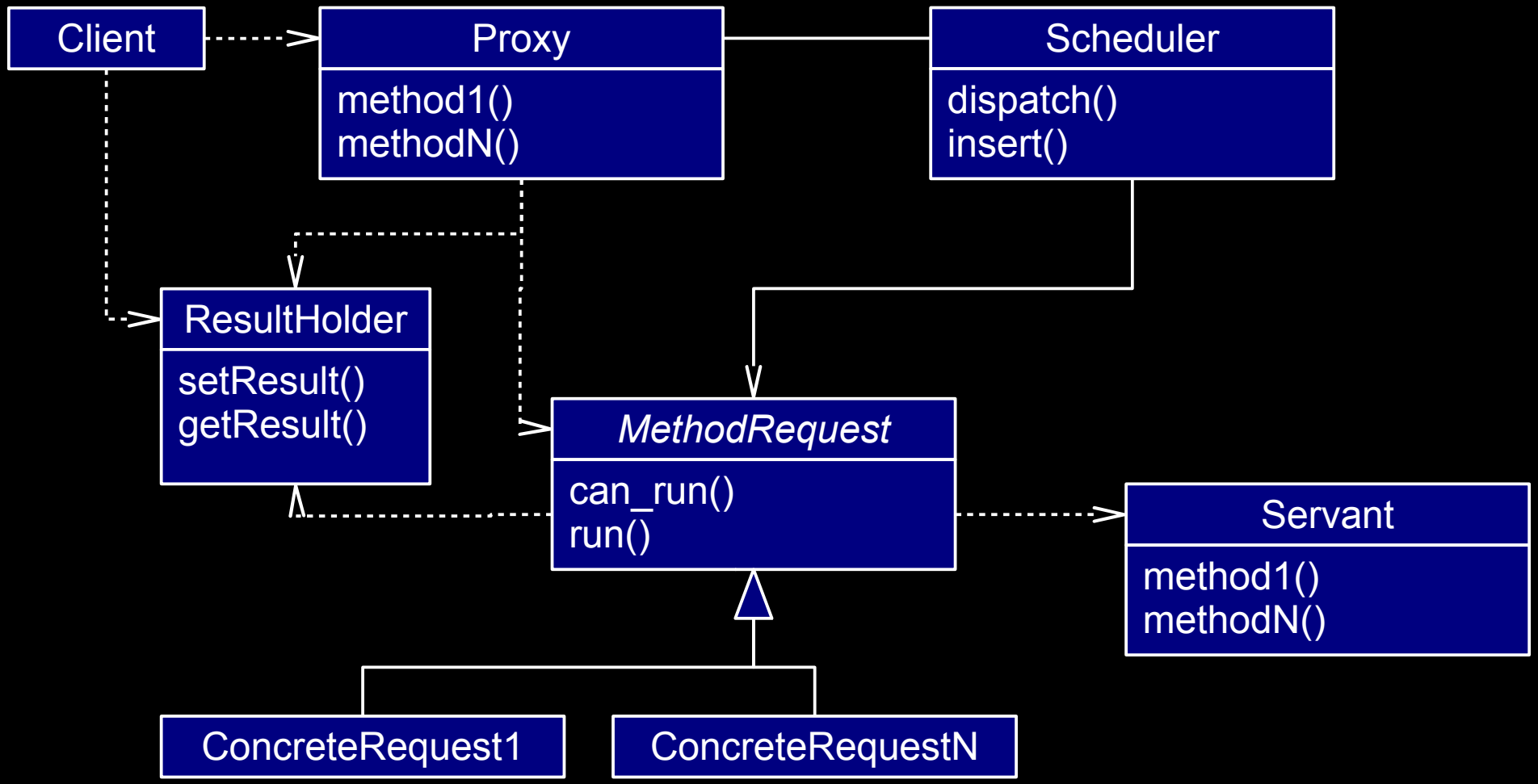
- Thread-safe Facade
  - tråd-säker ”Facade” – anropen synkroniseras där



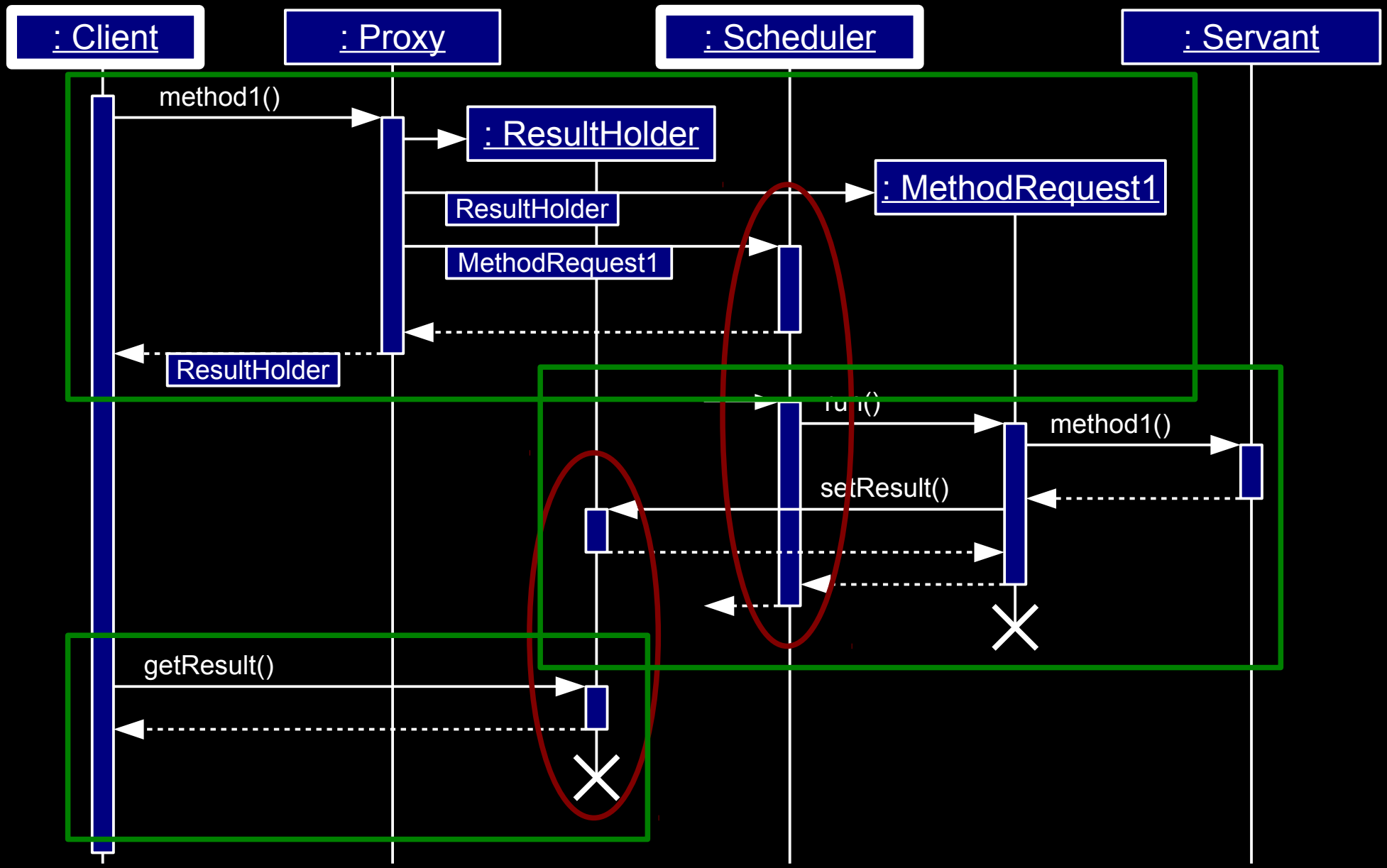
Active Object (Concurrent Object)

- Tillämpning
  - vill att anrop till CPU-intensiva metoder inte blockerar körning
  - vill att det ska vara lätt att synkronisera anrop till delade objekt
- Fördelar
  - förbättrad parallellism och förenklad synkronisering
  - beräknings-ordning kan vara annan än anrops-ordning
- Nackdelar
  - kan i vissa fall ge sämre prestanda
  - parallellism försvårar felsökning
  - ändrad beräknings-ordning försvårar felsökning

Active Object (Concurrent Object)



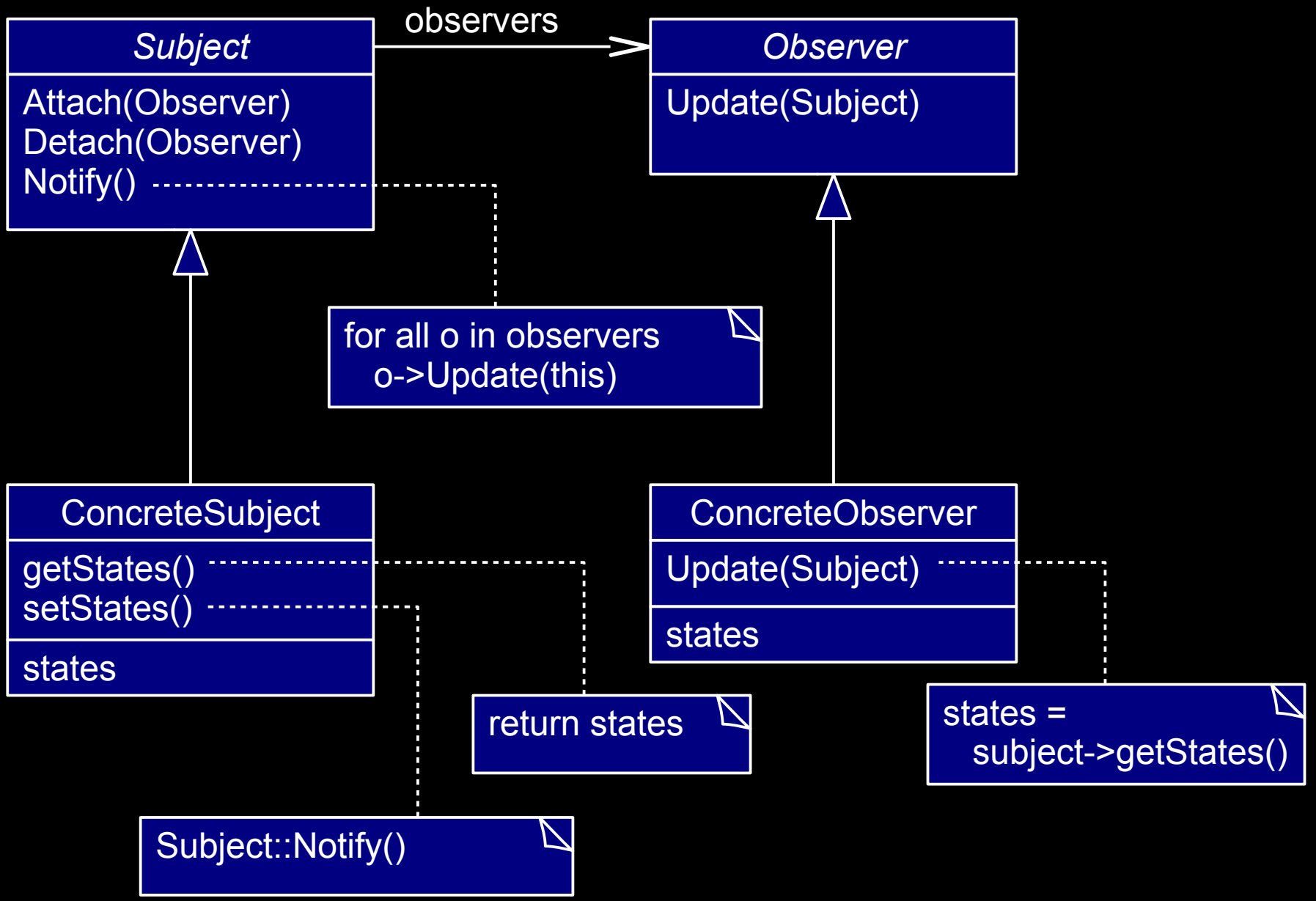
Active Object (Concurrent Object)



Observer (Publish-Subscribe)

- Tillämpning
  - när förändring i ett objekt leder till att andra behöver uppdateras
  - när en abstraktion har två aspekter, en beroende på den andra
    → kapsla in dessa aspekter och återanvänd dem separat
- Fördelar
  - abstrakt koppling mellan subjekt och observerare
  - stöd för broadcast, en meddelar flera lyssnare
- Nackdelar
  - oväntade uppdateringar kan leda till mänger av uppdateringar
  - serier av händelser kan leda till för många uppdateringar
  - källan till problem kan vara svåra att hitta

Observer (Publish-Subscribe)



Observer (Publish-Subscribe)