

TNM094 – Medietekniskt kandidatprojekt

Modellering  
för  
objektorienterad analys och design

# Varför OO

- Människor har begränsat minne
- Datorprogram är stora och komplicerade
  - Oftast kan man inte begränsa storleken
  - Begränsa den mängd man behöver förstå på en gång!
  - Skapa portabla klumpar!



```

...
    TCLAP::ValueArg<std::string> arg_viewer
        ("", "viewer", "Viewer position", false, "", "x,y,z");
    cmd.add(arg_screen_ll);
    cmd.add(arg_screen_ur);
    cmd.add(arg_screen_up);
    cmd.add(arg_viewer);

    TCLAP::SwitchArg arg_showcursor("", "show-cursor",
        "Do not hide the mouse cursor.", false);

    cmd.add(arg_showcursor);

#ifdef UTM50_ENABLE_GDAL
    TCLAP::MultiArg<std::string> arg_image("", "image",
        "Load and add a GeoTiff image", false, "file");

    cmd.add(arg_image);
#endif

    try {
        cmd.parse(argc, argv);
    } catch (TCLAP::ArgException &e) {
        std::cerr << "error: " << e.error() << " for arg " << e.argId() << std::endl;
    }

    bool debug = arg_debug.getValue();

    osg::ref_ptr<utm50_core::Scene> scene(new utm50_core::Scene);

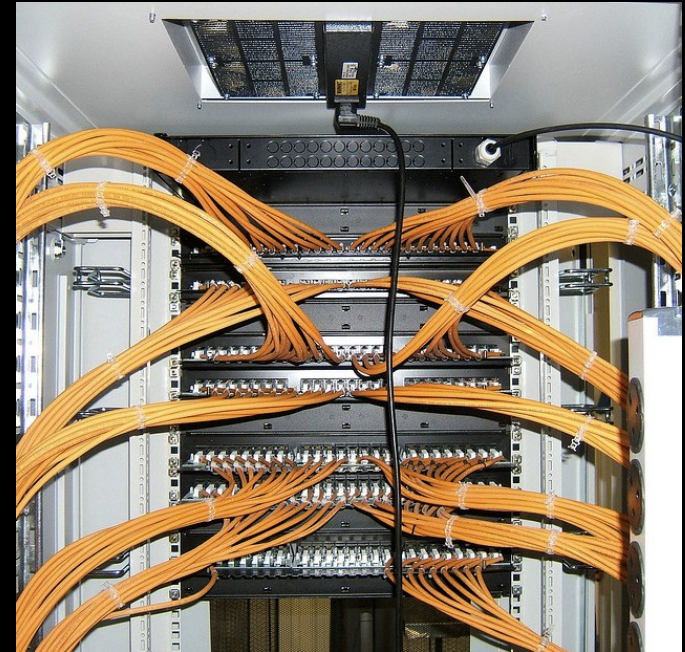
    {
        std::shared_ptr<utm50_core::TopViewNavigation> tvnav(new utm50_core::TopViewNavigation);
    }
...

```

```
int main(int argc, char *argv[]) {  
  
    CmdlineSettings conf;  
    conf.consumeCmdLineArguments(argc, argv);  
  
    Application app(conf);  
    app.run();  
  
    return app.getExitCode();  
}
```

# Varför OO

- Göm detaljer
  - kapsla in detaljer
  - visa upp enkla koncept
  - skydda från obehörig manipulation
- Definiera bra gränssnitt
  - gör det lätt att se sammanhang – vad som händer
  - gör det lättare att koppla samman delar



# Varför OO

- Objekt
  - Data som hör ihop
  - Operationer som hör ihop
- Vadå hör ihop?
  - Inkapslade koncept (t ex uppkoppling mot Facebook)
  - Data som hanteras samtidigt och måste stämma överens
  - Bäst: alla data och inga fler som behövs för konceptet (sammanhållning!)

# OOAD

- Analys
  - Hitta objekt-liknande koncept i problemformuleringen
  - Analysera problem och förutsättningar
- Design
  - Skapa en passade objekt-orienterad design
  - Gå från teori till praktisk tillämpning

# Objekt-orienterad analys

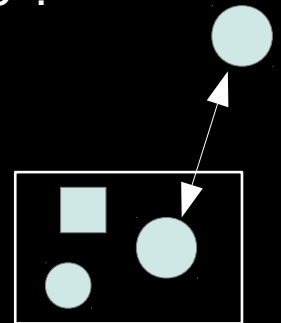
- Formalisera problembeskrivningen i objekt-form
- Identifiera vilka *entiteter* och *data* som hanteras
  - substantiv
  - *boll* och *ring*
- Identifiera vad de *gör* eller vad som *görs* med dessa
  - verb
  - *se* och *höra*

*“Programvaran ska koppla upp sig mot Facebook och hämta hem alla bilder relaterade till den inloggade användaren.”*



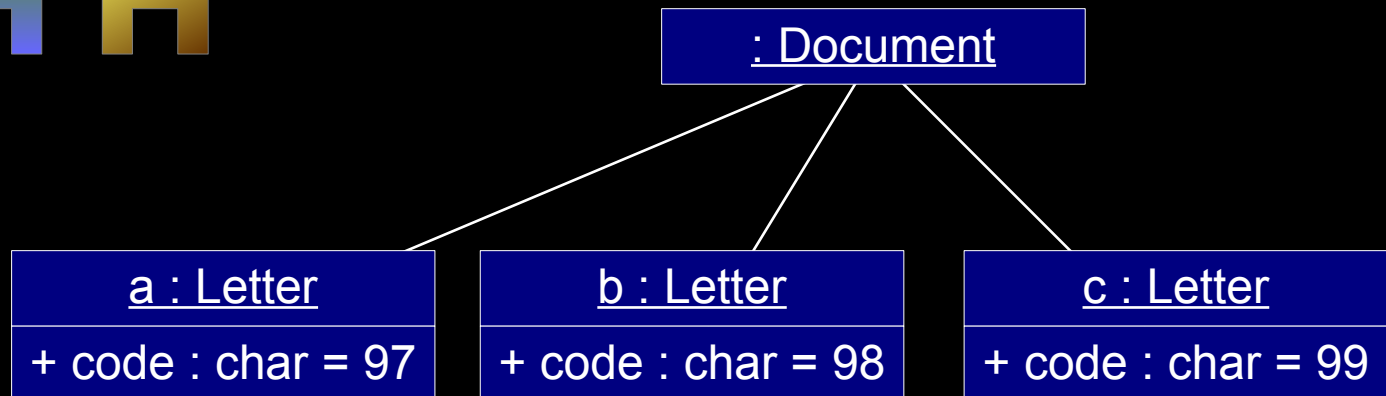
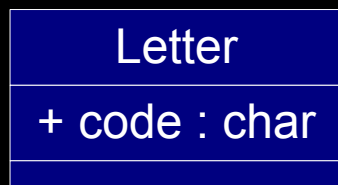
# Objekt-orienterad analys

- Vad är ett objekt?
  - Entiteter, deras data och funktioner – realisering av koncept
  - Använd verkliga koncept eller domänspecifika termer
- Hur väljer vi våra objekt?
  - Är det rätt koncept vi vill hantera?
    - "Facebook" eller mer generellt koncept "Media" eller "Service"?
    - Tänk om vi vill hämta från "Facebook och Instagram"?
  - Hur förhåller sig våra koncept till varann?
    - Är "Facebook" en bild som hämtas via Internet?  
eller en lista med bilder? eller en lista med media?  
eller kommer "Facebook" *ha* en lista med bilder?
    - Kommer "Service" vara en del av vårt system?  
eller är "ServiceConnection" det koncept vi vill arbeta med?

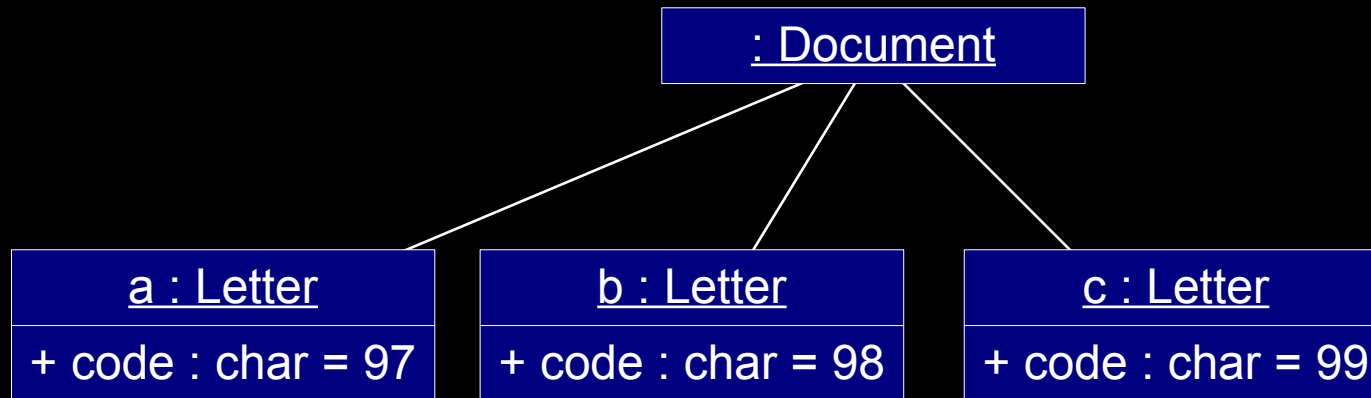


# Modellering av objekt och klasser

- UML – Unified Modelling Language
  - grafiskt språk med strikt semantik
- Klass-diagram
  - beskriver klasser och deras innehåll
  - kan även beskriva instanser
  - beskriver hur de förhåller sig till varandra



# Exempel



```
class Document {
public:

    void printDocument();

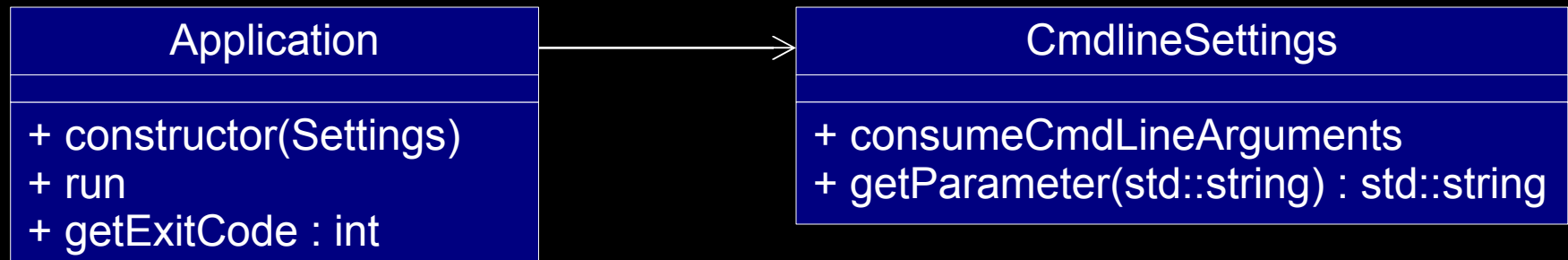
private:
    std::vector<Letter> text;
};
```

# Objekt-orienterad design

- Gå från analys till design
  - Vilka objekt bör vara objekt i programmet?
  - Vilka dolda/underförstådda objekt måste vi ta hänsyn till?
  - Behöver vi fler, stödjande objekt?
  - Hur förhåller sig våra objekt till varann?
  - Kan vi generalisera för återanvändbarhet?

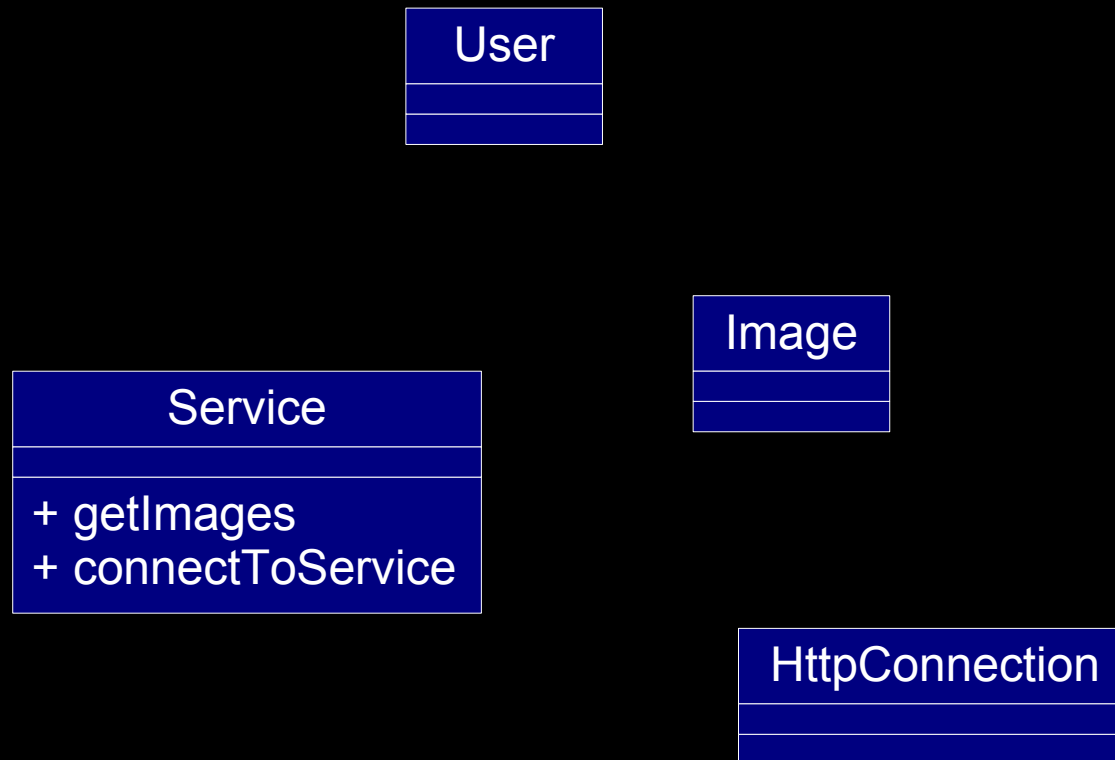
# Exempel

```
int main(int argc, char *argv[]) {  
  
    CmdlineSettings conf;  
    conf.consumeCmdLineArguments(argc, argv);  
  
    Application app(conf);  
    app.run();  
  
    return app.getExitCode();  
}
```



# Exempel

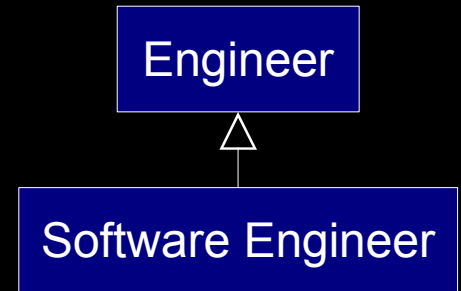
*“Programvaran ska koppla upp sig mot Facebook och hämta hem alla bilder relaterade till den inloggade användaren.”*



# Arv eller Komposition

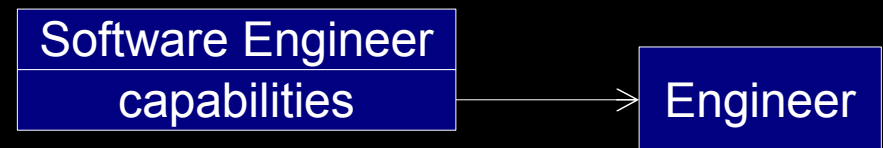
- Arv (is-a)

- statiskt kompilerad struktur
- direkt-access till medlemsvariabler
- möjlighet till polymorfism



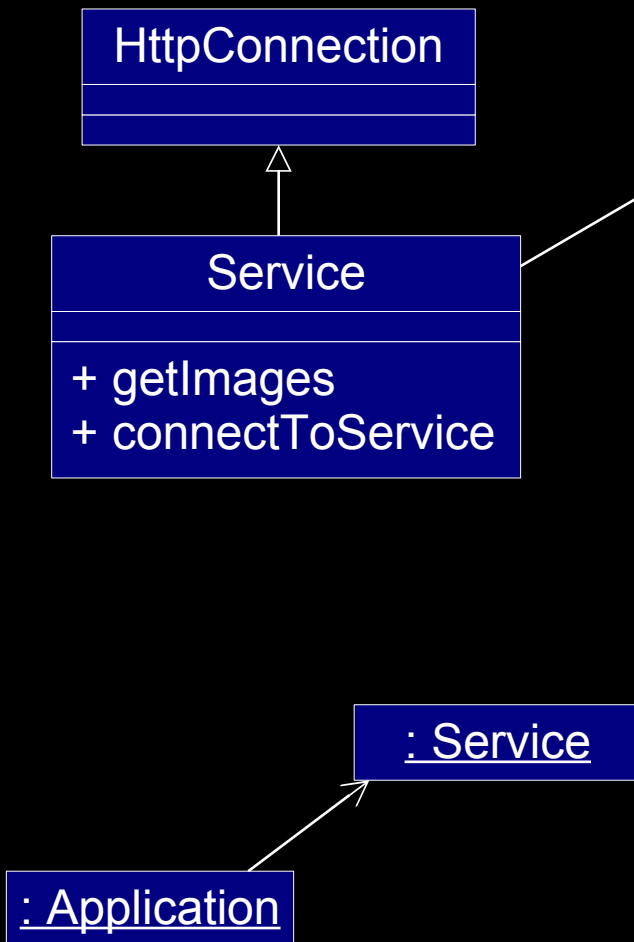
- Komposition (has-a)

- mer flexibel – strukturen kan ändras under körning
- därför också svårare att visualisera och resonera kring
- i många fall mer naturlig struktur än arv



# Exempel

*“Programvaran ska koppla upp sig mot Facebook och hämta hem alla bilder relaterade till den inloggade användaren.”*



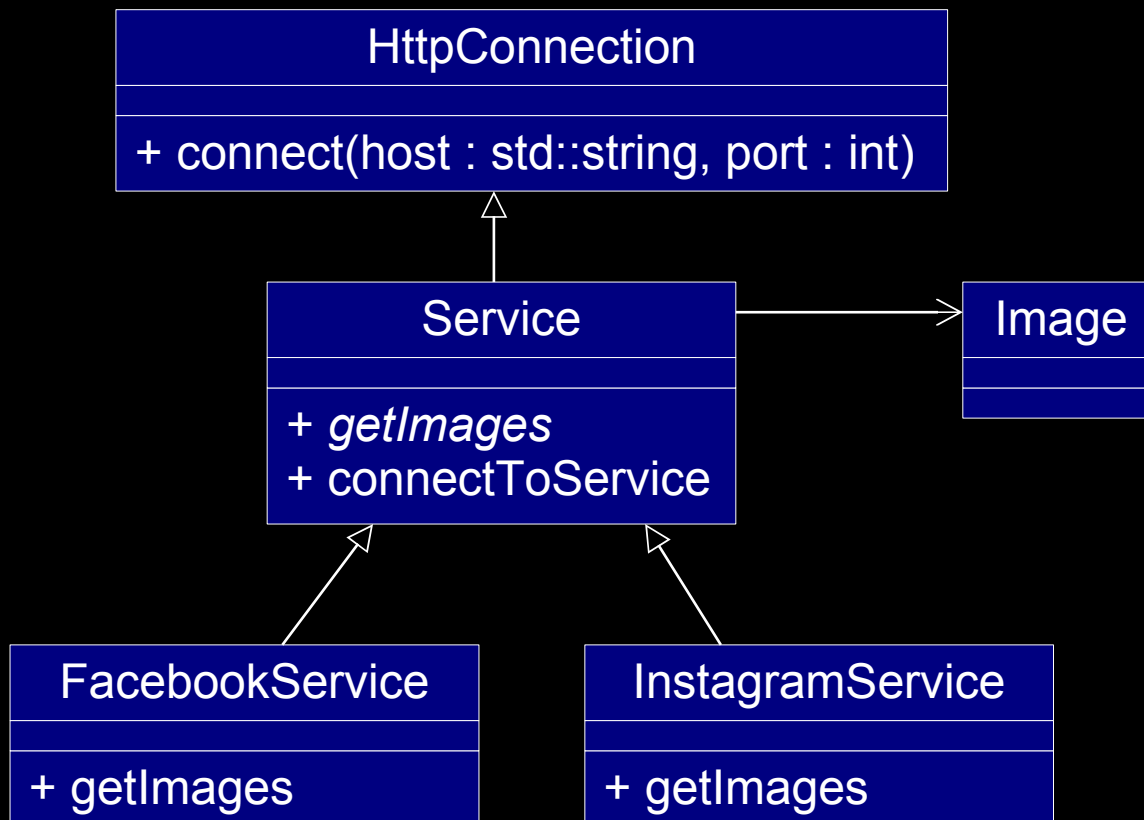
```
class HttpConnection {  
    ...  
    void connect(User);  
    ...  
};
```

```
class Service : public HttpConnection {  
    ...  
    bool getImages(std::vector<Image> &img);  
    ...  
};
```



# Både Facebook och Instagram

- Samma koncept, men olika implementation
  - Abstraktion och utbyggnad



# Abstraktion

”Allmänna begrepp härledda från specifika exempel”

- Hantera olika implementationer på samma sätt
  - Ha lista av Service-objekt oavsett implementation
- Separera gränssnitt och implementation
  - Definiera ett (eller flera) enkla gränssnitt
  - Utelämna oviktiga detaljer
  - En implementation kan följa detta gränssnitt

# Exempel

```
struct ImageServiceInterface {  
    bool getImages(std::vector<Image>&) = 0;  
};
```

```
struct VideoServiceInterface {  
    bool getVideos(std::vector<Video>&) = 0;  
};
```

...



```
class FacebookService  
    : public ImageServiceInterface,  
      public VideoServiceInterface {  
  
    bool getImages(std::vector<Image>&);  
    bool getVideos(std::vector<Video>&);  
  
    ...  
};
```

# Gränssnitt (interfaces)

- Kopplingen från en enhet till en annan
  - håll gränssnittet stabilt för att undvika spridning av förändring
  - undvik optimering genom gränssnitt (algorithm-specifika gränssnitt)
- Interface Segregation Principle (ISP)
  - en enhet kan ha flera gränssnitt för olika syften
  - dela upp stora gränssnitt i flera mindre och mer specifika
- Specifikation i dokumentationen
  - syfte – vad används gränssnittet (klassen/funktionen) till
  - preconditions (förutsättningar) – internt tillstånd, globala resurser, etc. 
  - protokoll – hur ska gränssnittet användas
  - postconditions (resultat) – interna effekter, ändringar och returnerade data
  - kvalitetsattribut – prestanda, pålitlighet, trådsäkerhet, etc.

# Dölja information

- Syfte

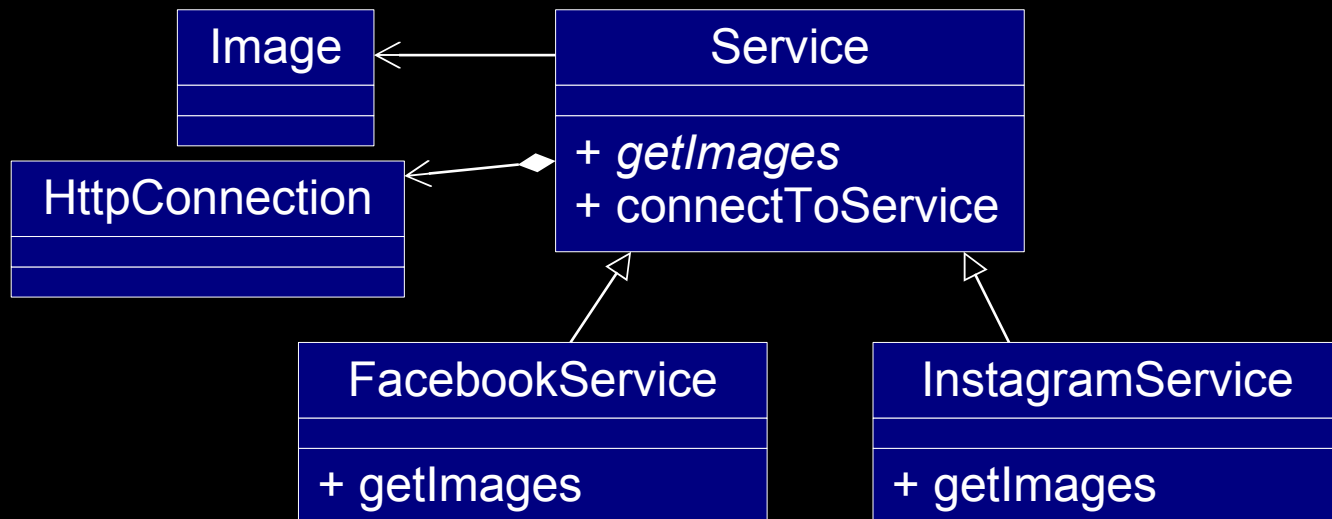
- kapsla in förändringsbenägna design-val 
- gömma interna data-representationer 
- gömma vilken algoritm som används
- erbjuda ett stabilt gränssnitt

- Trade-off

- + inkapslad kod och lättbegripliga moduler
- + enkelt att hitta orsak och verkan
- data kan behöva konverteras
- stort antal små moduler

# Både Facebook och Instagram

- Försök att använda arv smart
  - *Inte* för att få tillgång till funktionalitet
  - För att *abstrahera* och återanvända
  - För att *bygga ut* funktionalitet
- Annat genom komposition (has-a)
  - För att återanvända extern funktionalitet



# Exempel

```
class Service {  
  
    bool connect(std::string host, int port) {  
        connection.reset(new HttpConnection(host, port));  
        if (!connection.isConnected()) {  
            connection.reset(nullptr);  
        }  
        ...  
    }  
  
    virtual bool getImages() = 0;  
    ...  
  
protected:  
    std::unique_ptr<HttpConnection> connection;  
};
```

# Utbytbarhet (substitutability)

- Att använda subklass istället för basklassen
- Liskov Substitutability Principle (LSP)
  - subklassen stödjer alla metoder
  - metodernas signatur är kompatibel
    - förutsättningar är identiska, eller lägre krav
    - resultaten är identiska, eller mer som händer
- Inte en regel men en bra princip
  - när behöver man hantera subklasser explicit
  - när kan man byta ut en klass mot valfri subklass



# Generalisering

- Gör moduler mer generellt applicerbara
  - ökad återanvändning
  - minskad underhållskostnad
  - `std::sort`
- Genom att
  - ta emot kontext-specifik information som argument
  - minska krav på förutsättningar
  - minska eller förenkla resultaten
- Trade-off
  - prestanda, användbarhet och utvecklingstid
  - över-generaliserad kod är dyr

# Kvalitet hos klasser och objekt

- Samstämmighet
  - Sammanhållning – cohesion!
  - Single Responsibility Principle (SRP)
    - varje klass ska bara ha *en uppgift*  
uppgiften ska *inte delas* med andra klasser  
*men* vi kan *använda* andra klasser för att lösa uppgiften
    - varje klass har bara *en anledning att ändras*  
(om uppgiften ändras)
- Koppling
  - Objekt samarbetar mycket med andra objekt
  - Se till att samarbetet inte är *för* intimt!

# Varför modellera?

- För dokumentation
  - process-modellering
  - kravspecifikation
  - system-arkitektur och programdesign
- För förståelse
  - struktur, beroenden, uppgifter och ansvar
  - utforska händelser, sekvenser, timing, race conditions, etc.
- För att kommunicera
  - i diskussioner på whiteboard eller via dokument
  - beskriva tankar och idéer

# Vanliga UML-diagram

- Krav och problembeskrivning
  - Vad ska systemet göra?
    - **Use case-diagram**
    - **Aktivitetsdiagram**
- Systemets delar
  - Vilka delar ska göra detta?
    - **Komponent-diagram**
    - **Klassdiagram**
- Implementation
  - Hur ska delarna samarbeta?
    - **Tillståndsdigram** – visar interna tillstånd
    - **Samarbetsdiagram** – visar anropsstruktur
    - **Sekvensdiagram** – visar anropsordning

Abstrakt  
(Hög nivå)  
Vad

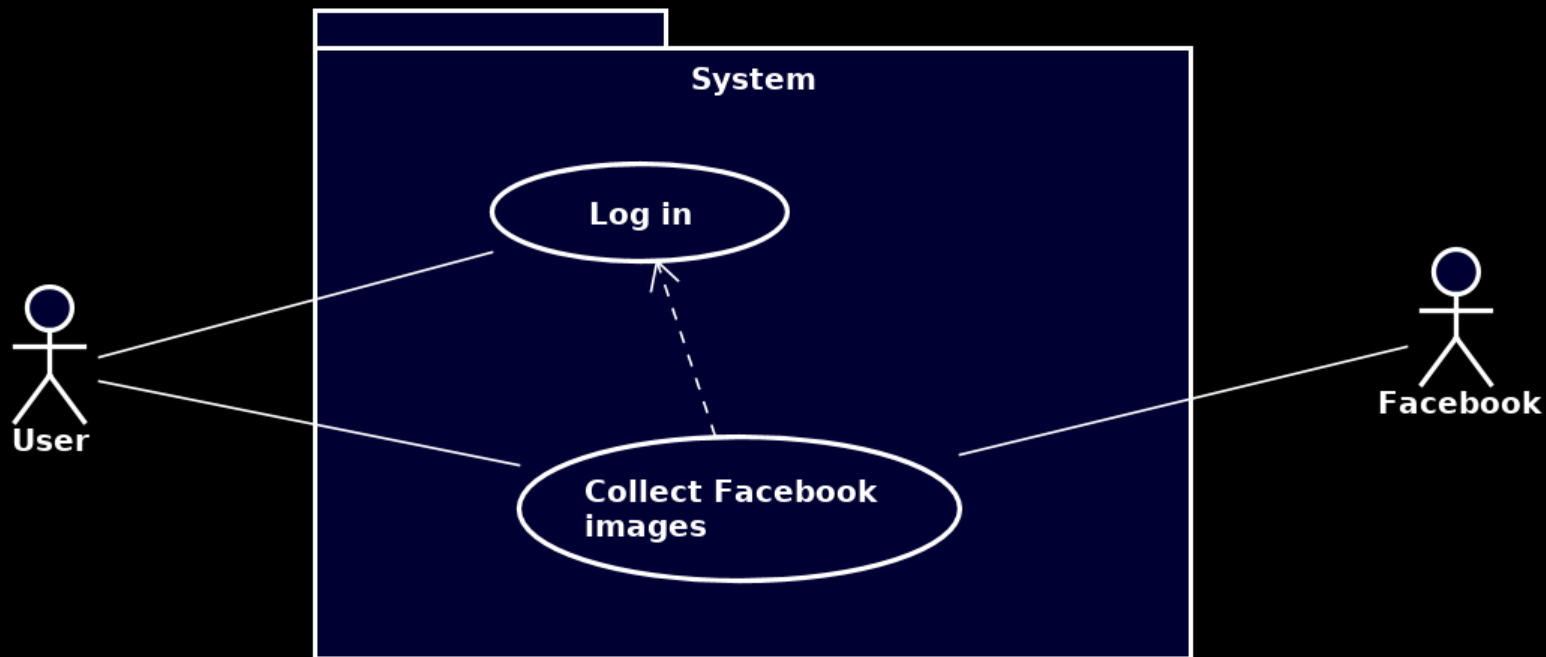


Konkret  
(Implementation)  
Hur

# Use case-diagram

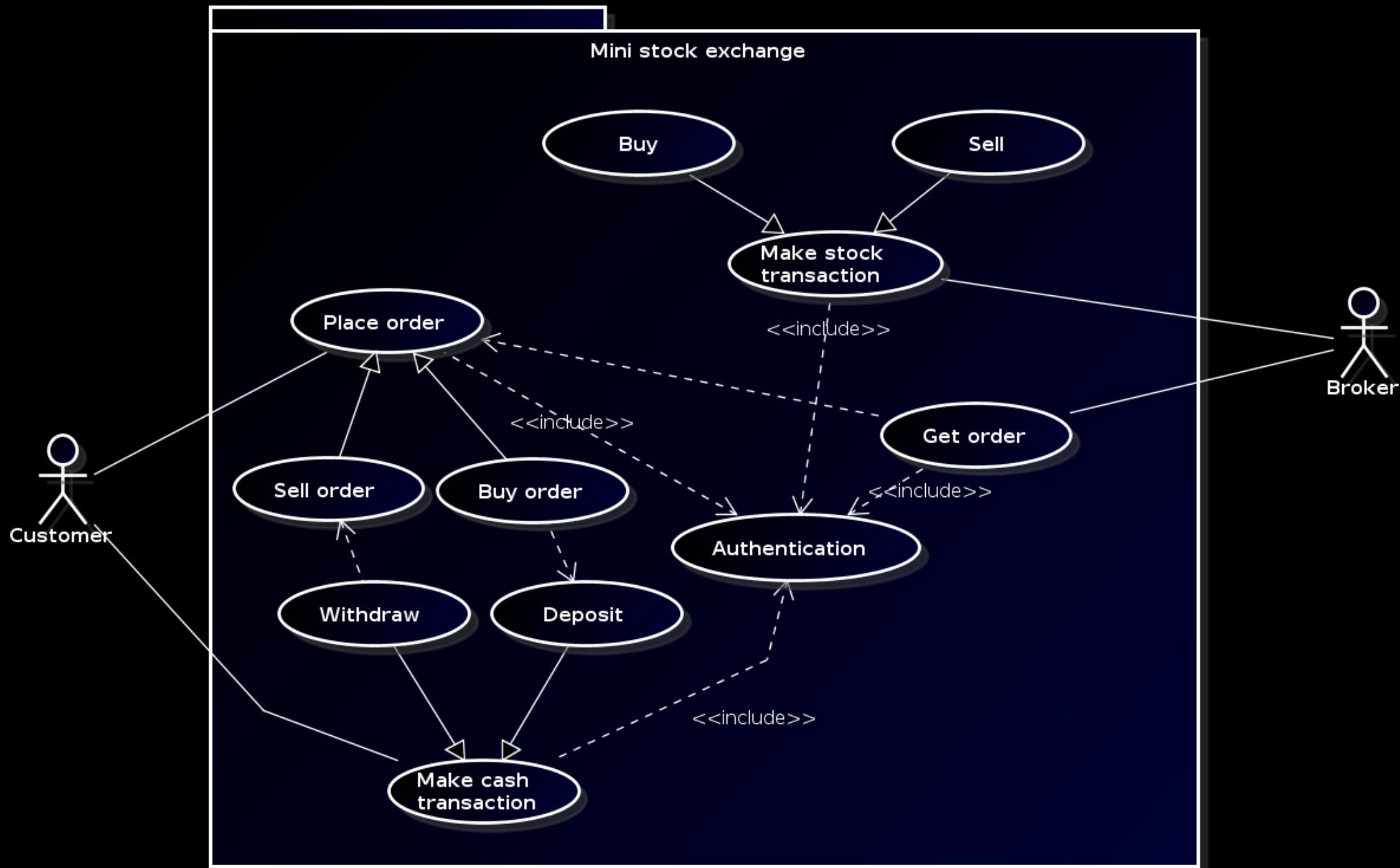
- Vad ska systemet göra

*“Programvaran ska koppla upp sig mot Facebook och hämta hem alla bilder relaterade till den inloggade användaren.”*



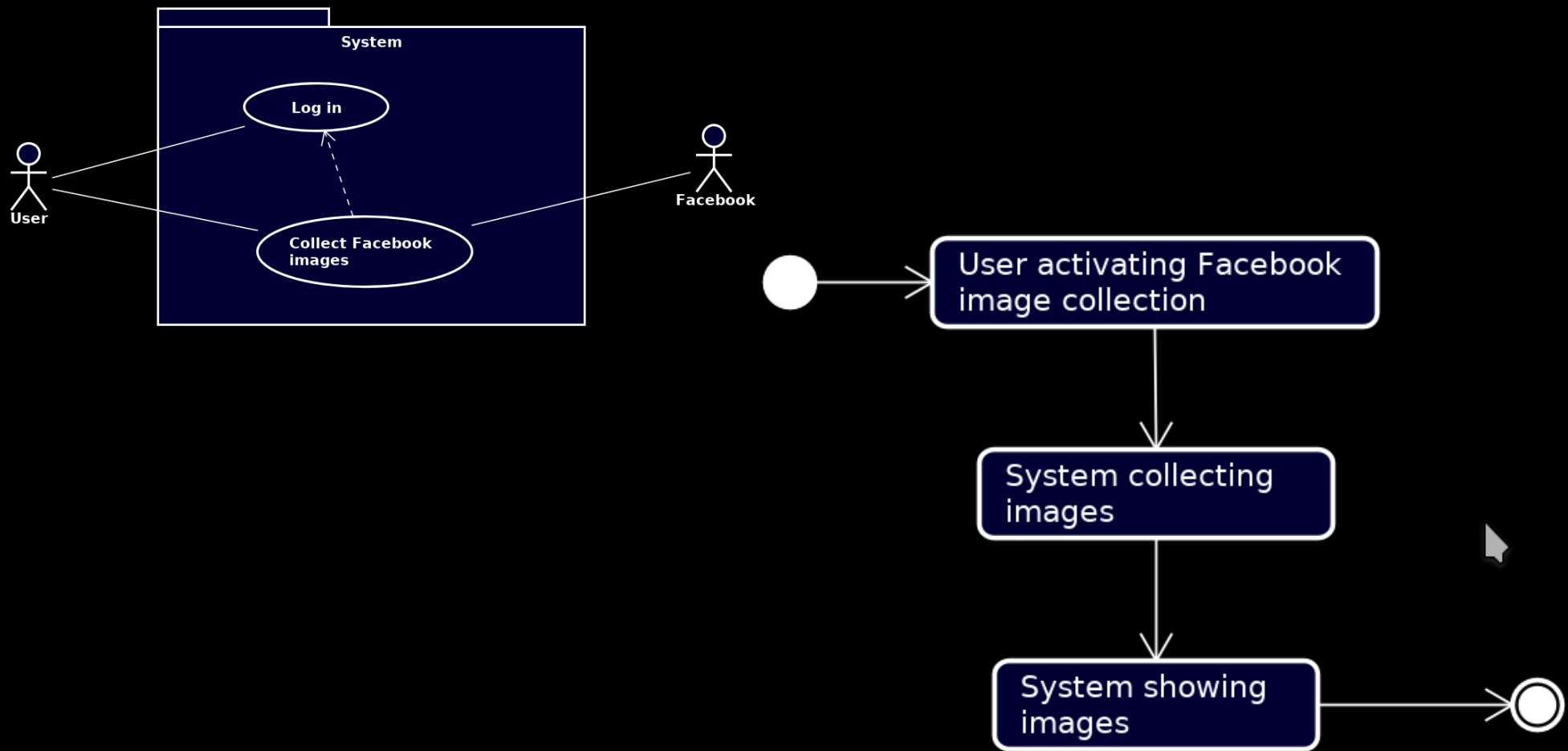
# Use case-diagram

- Syfte
  - Definiera och strukturera användningen av ett system
  - Markera gränsen för systemet och andra resurser
- Komponenter
  - Aktörer (actors)
    - Vem interagerar med systemet (vänster sida)
    - Vilka andra system interagerar systemet med (höger sida)
  - Use case
    - Vad man kan göra med systemet
    - Abstraktion av beteendet
    - Följdbehov från en direkt användning



# Aktivitetsdiagram

- Vad ska systemet göra (i detalj)?
  - abstrakt beskrivning – inte implementation!

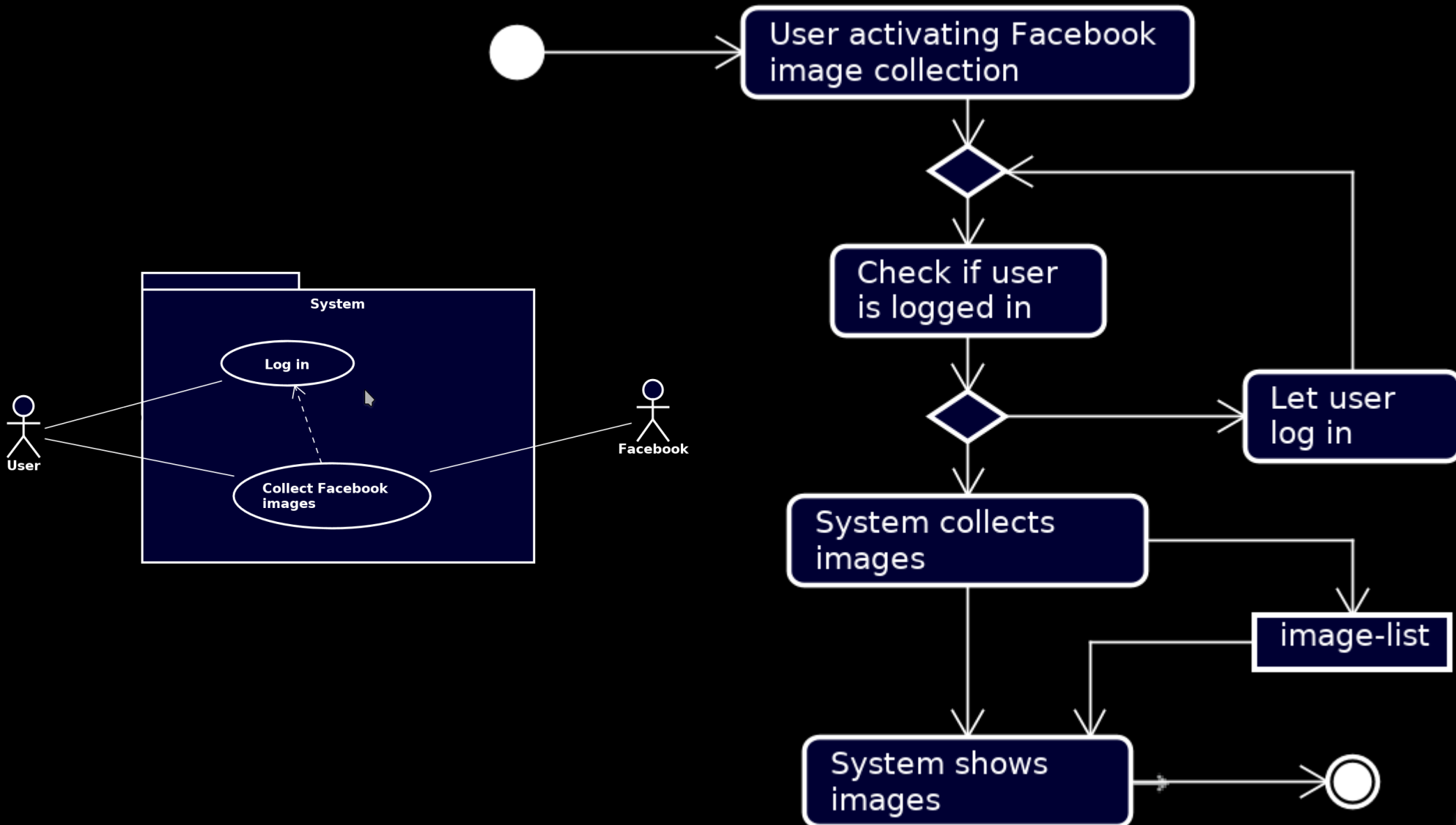




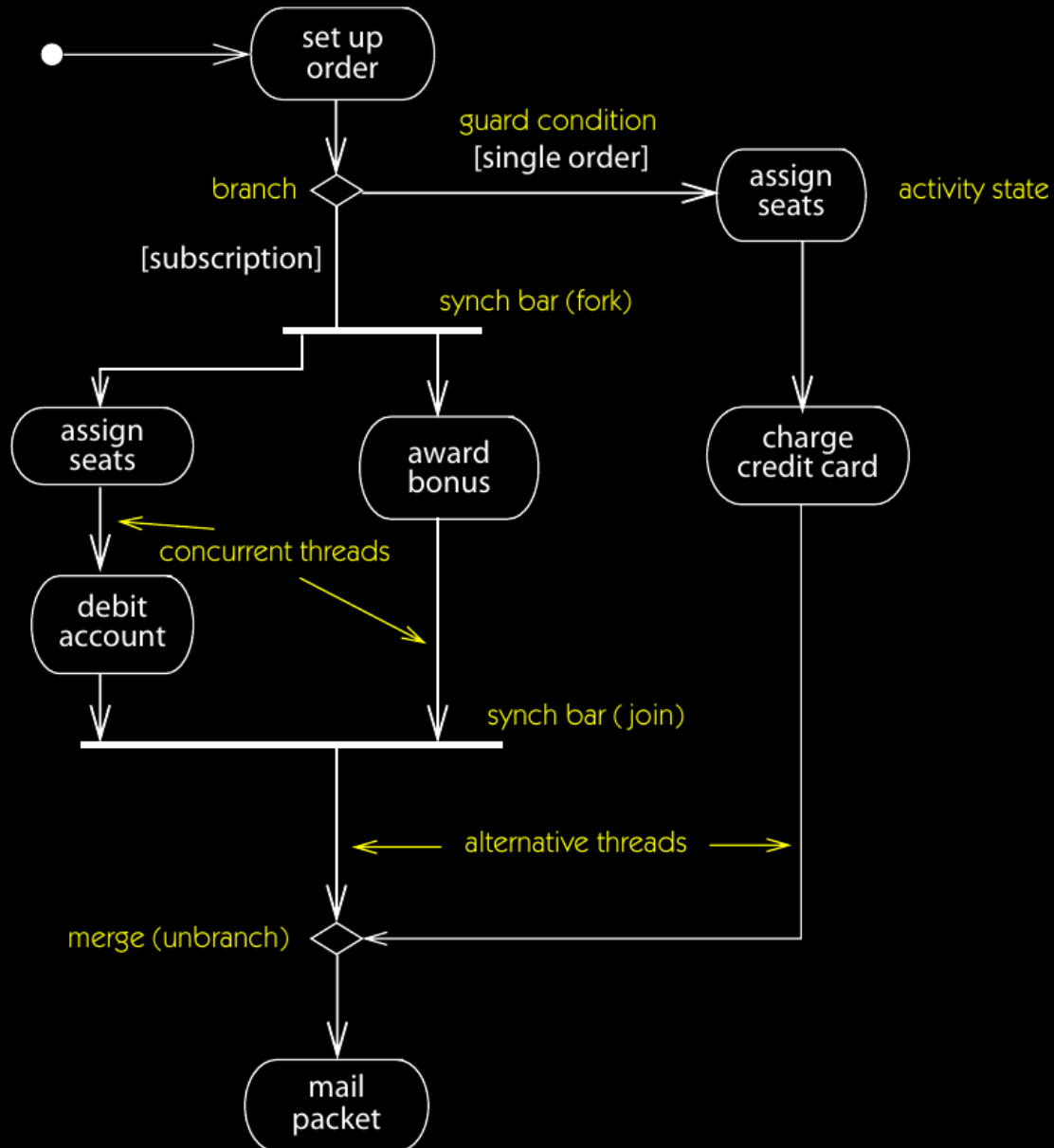
# Aktivitetsdiagram

- Beskriver en process på abstrakt nivå
  - Vad ska hända?
  - I vilken ordning måste sakerna hända?
  - Vad kan ske samtidigt?
  - Vad ska hända om...?
- Viktiga delar
  - Början och slut
  - Aktiviteter och objekt
  - Övergångar från en aktivitet till en annan
  - Villkorade och parallelliserande förgreningar
  - Uppdelning i banor för processande enheter (swimlanes)

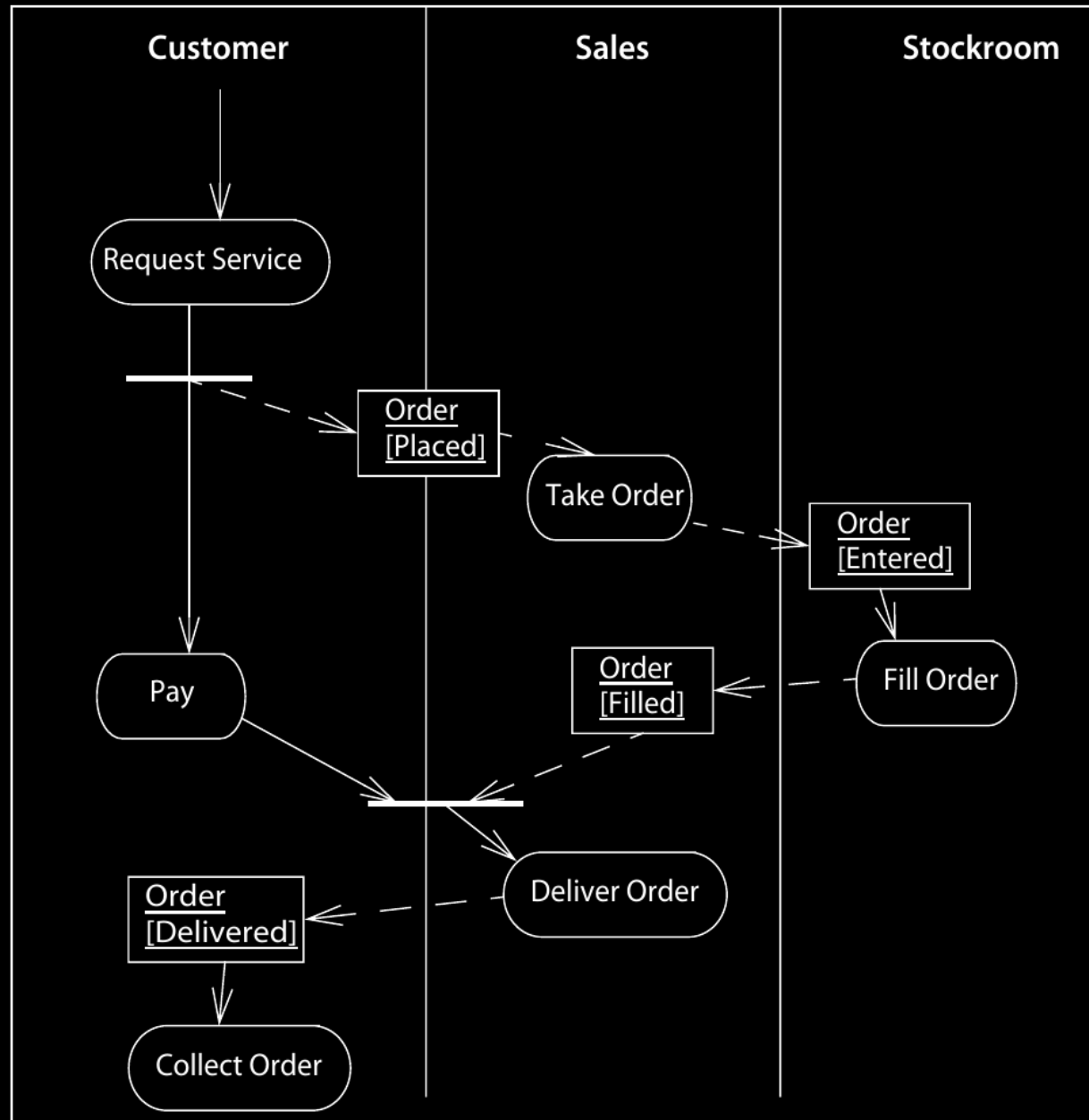
# Aktivitetsdiagram



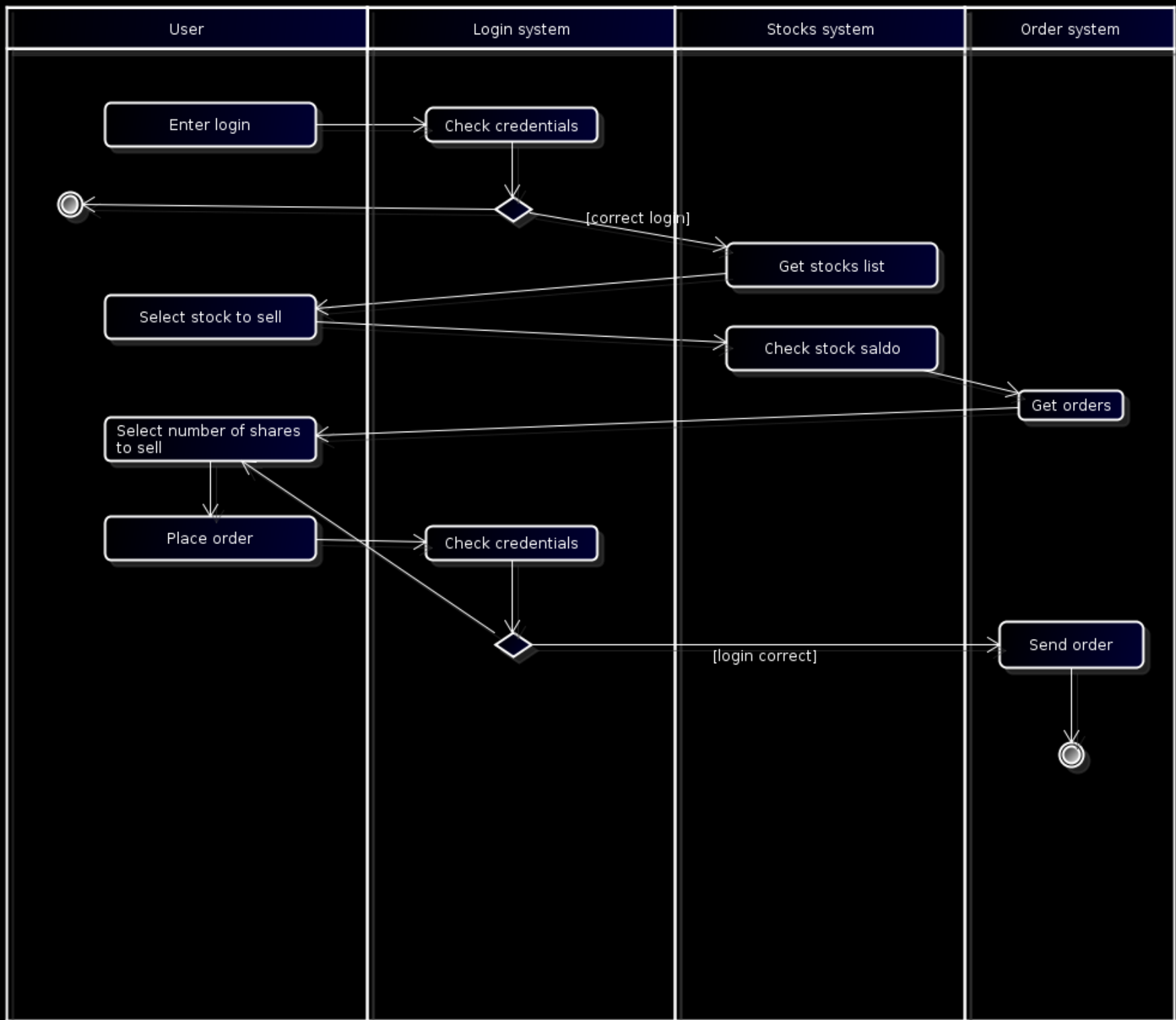
# Aktivitetsdiagram



# Aktivitetsdiagram



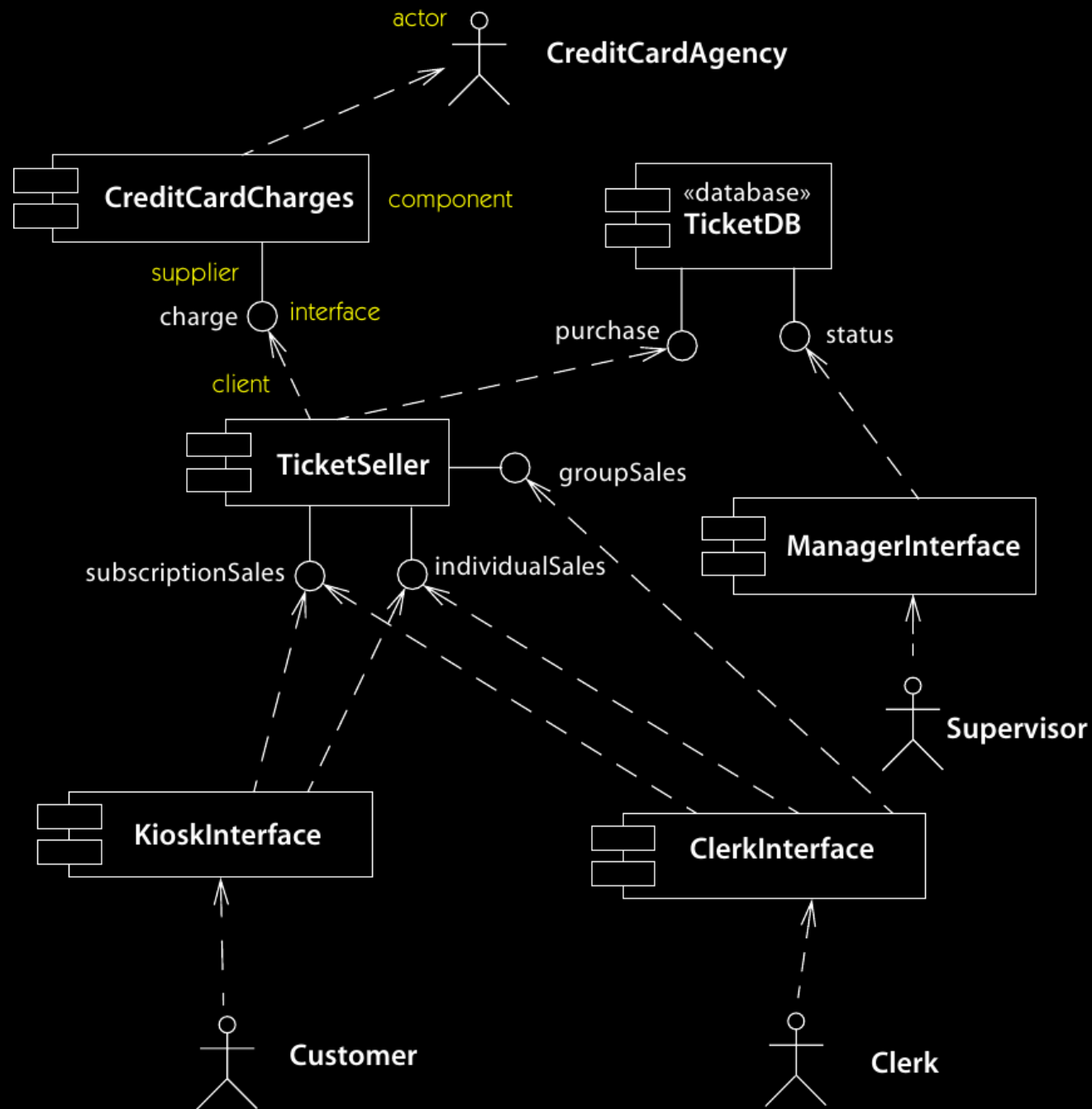




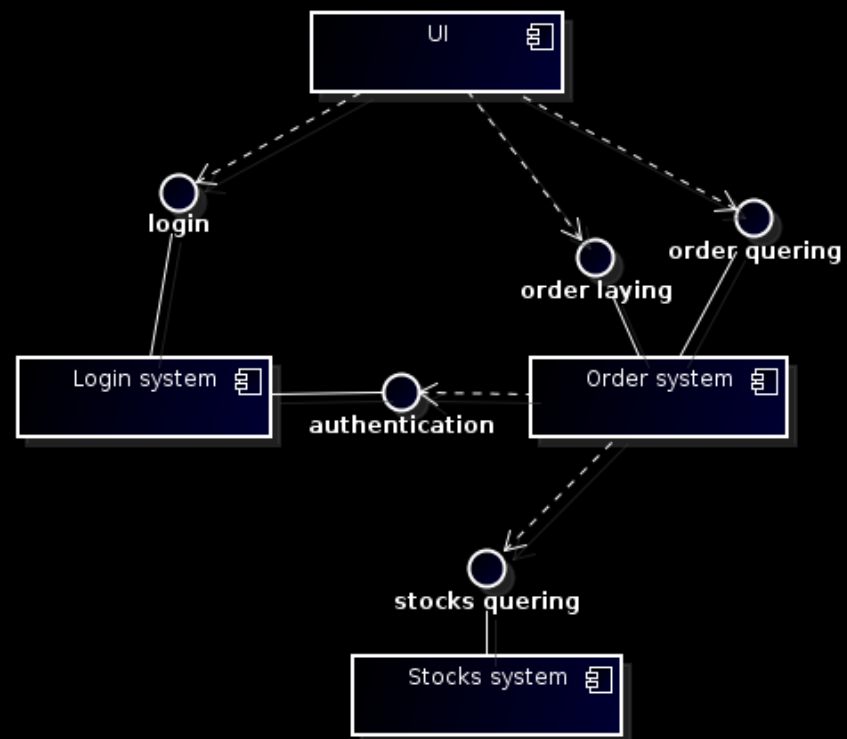
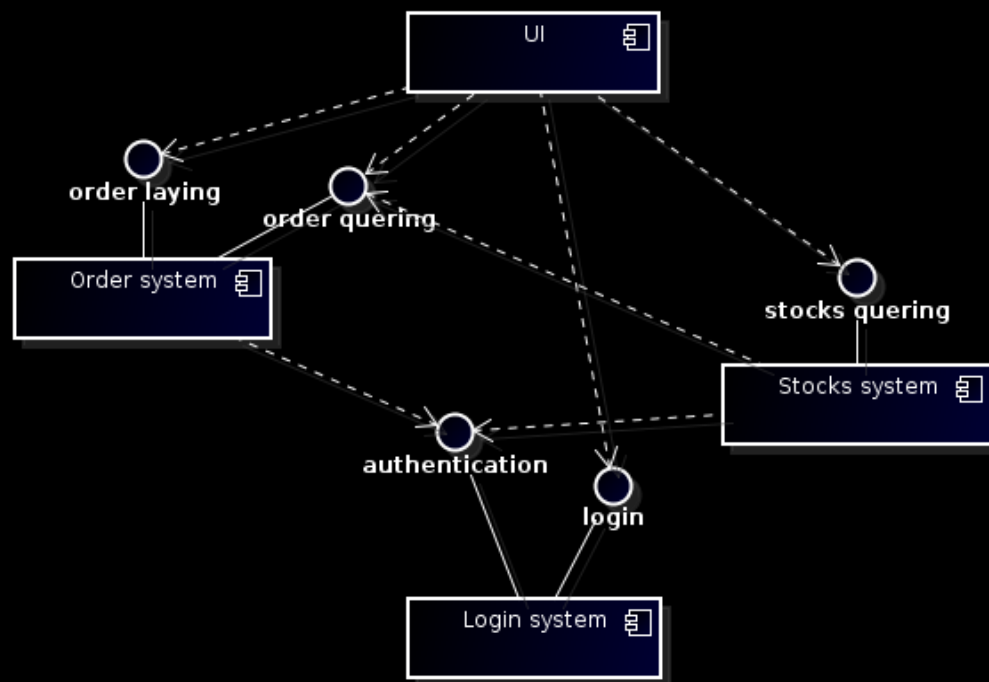
# Komponentdiagram

- Syfte
  - Strukturera systemet på abstrakt nivå
- Viktiga delar
  - Aktörer och komponenter
  - Gränssnitt
  - Beroenden

# Komponentendiagram







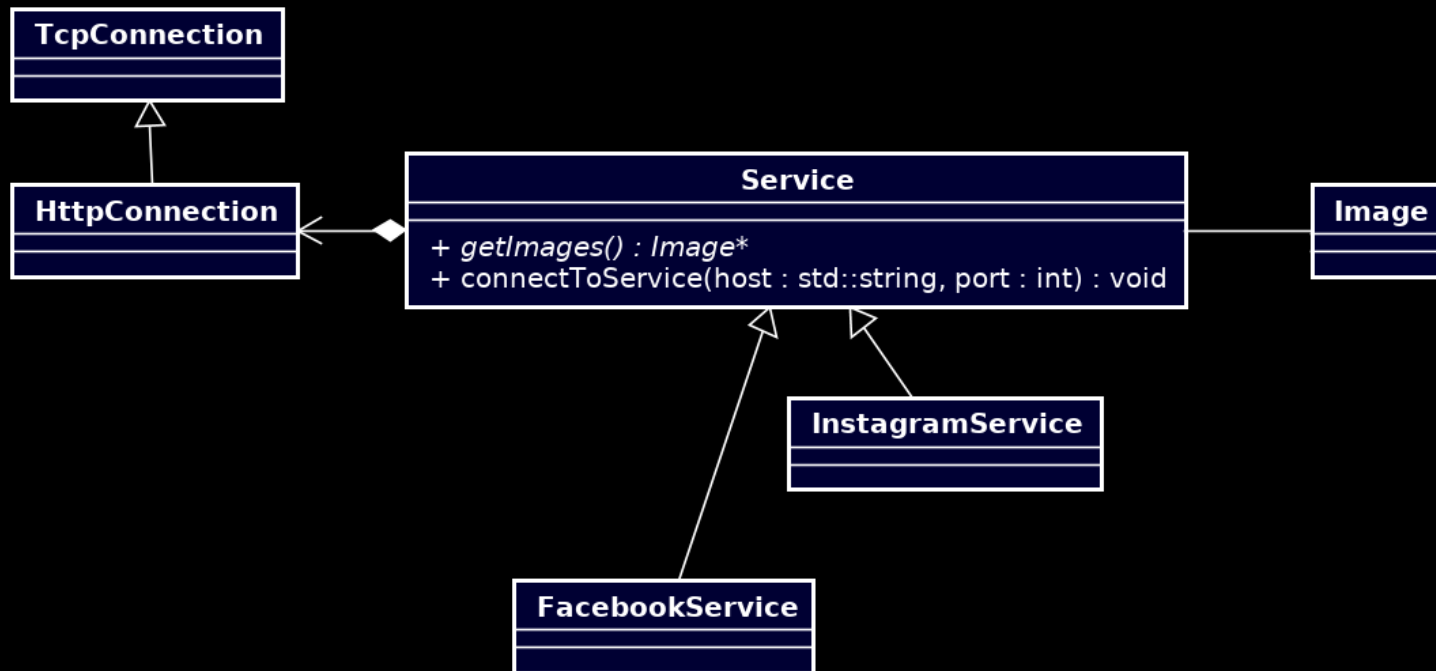
# Klassdiagram

- Syfte
  - Strukturell vy – beskriva systemet struktur
  - Viktigast i programdesign
  - Användbar för modellering av systemarkitekturer
- Viktiga delar
  - Klasser, variabler och metoder
  - Beroenden och arv

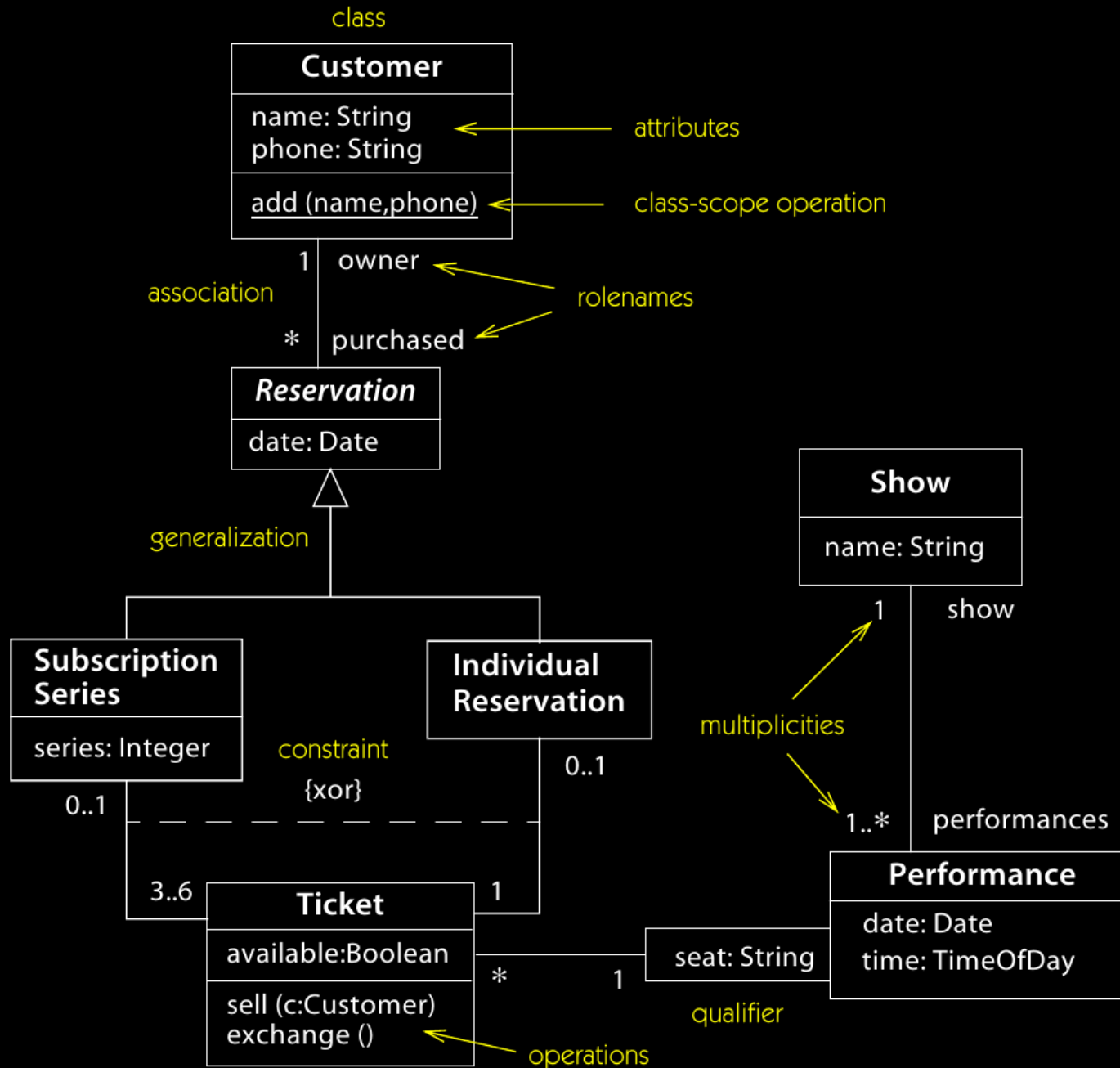
men också

  - Paket – strukturera på en abstrakt nivå
  - Objekt – visa på samarbeten vid körning

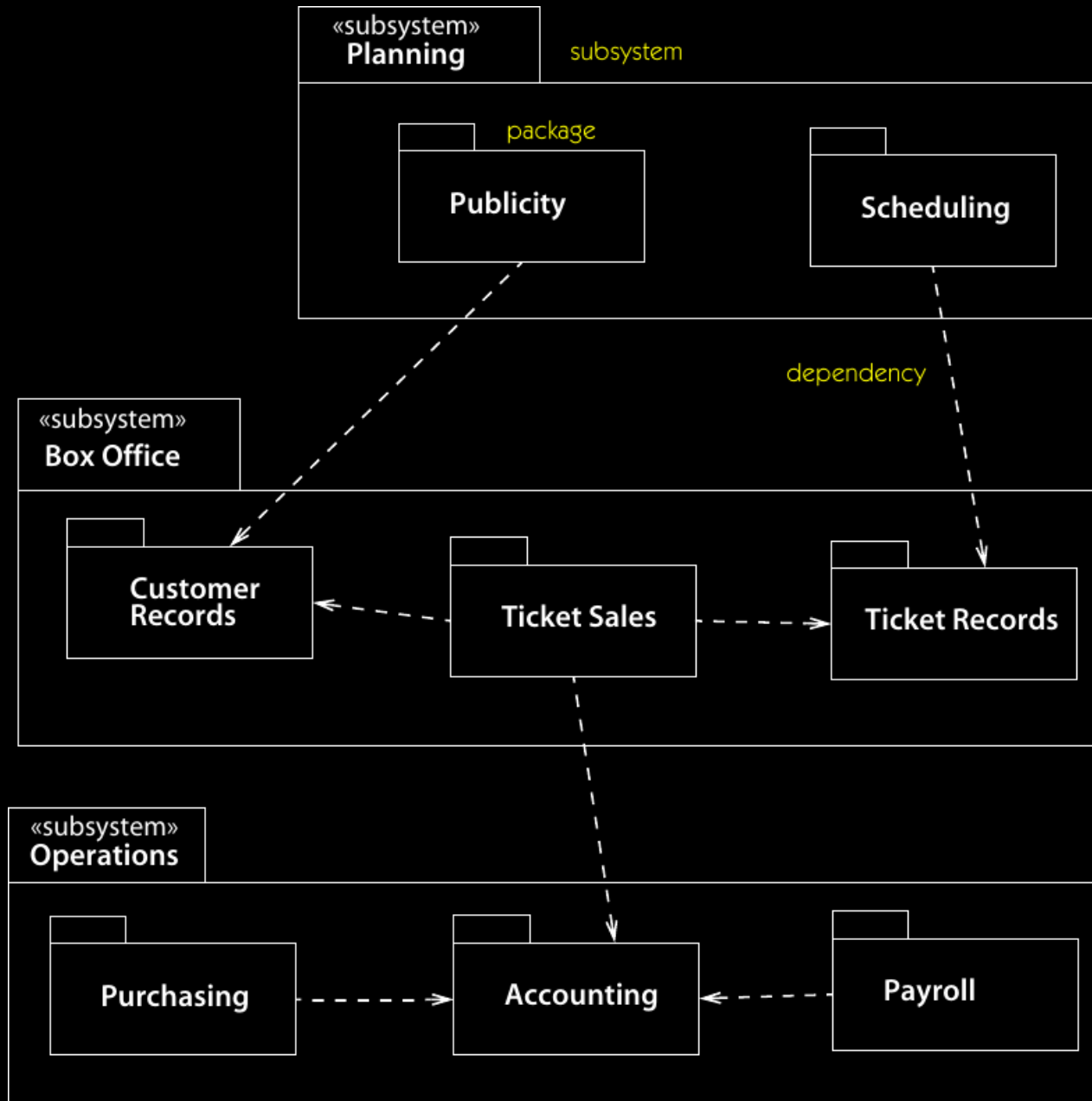
# Klassdiagram



# Klassdiagram

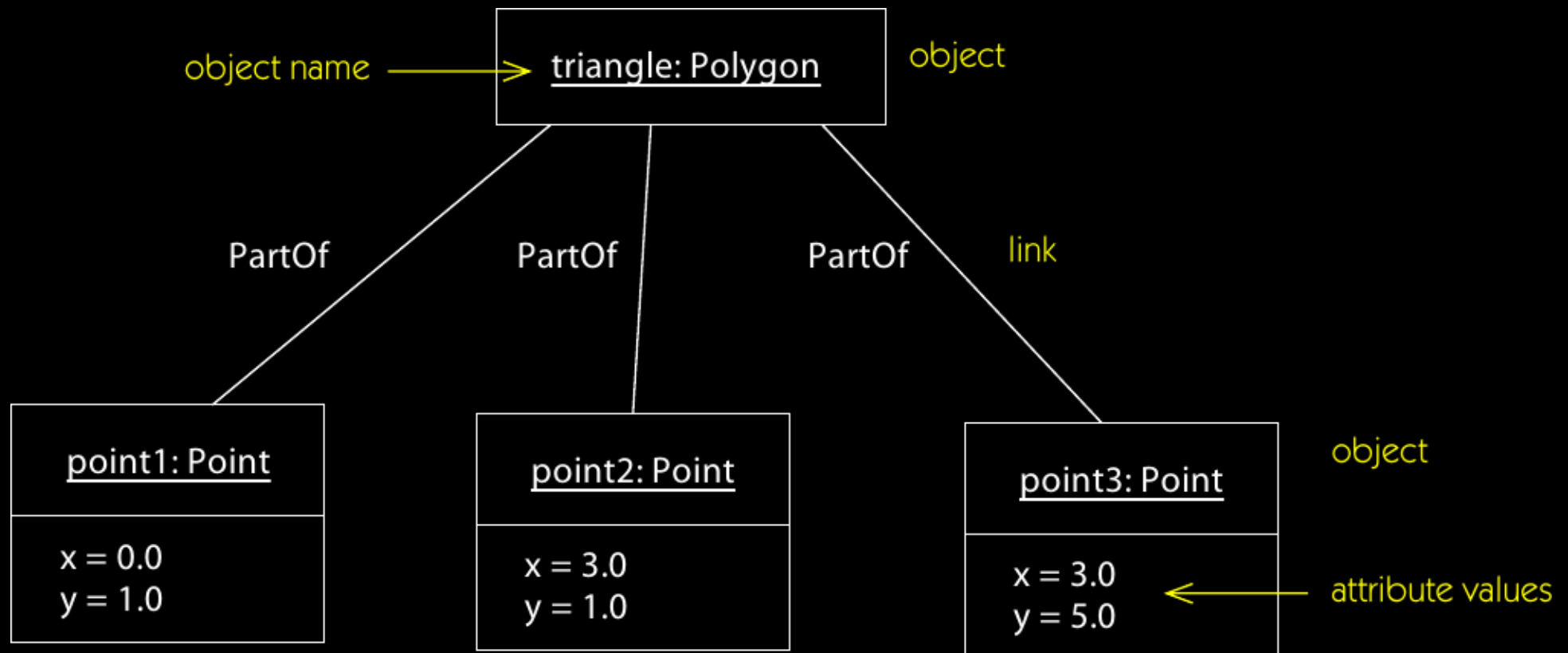


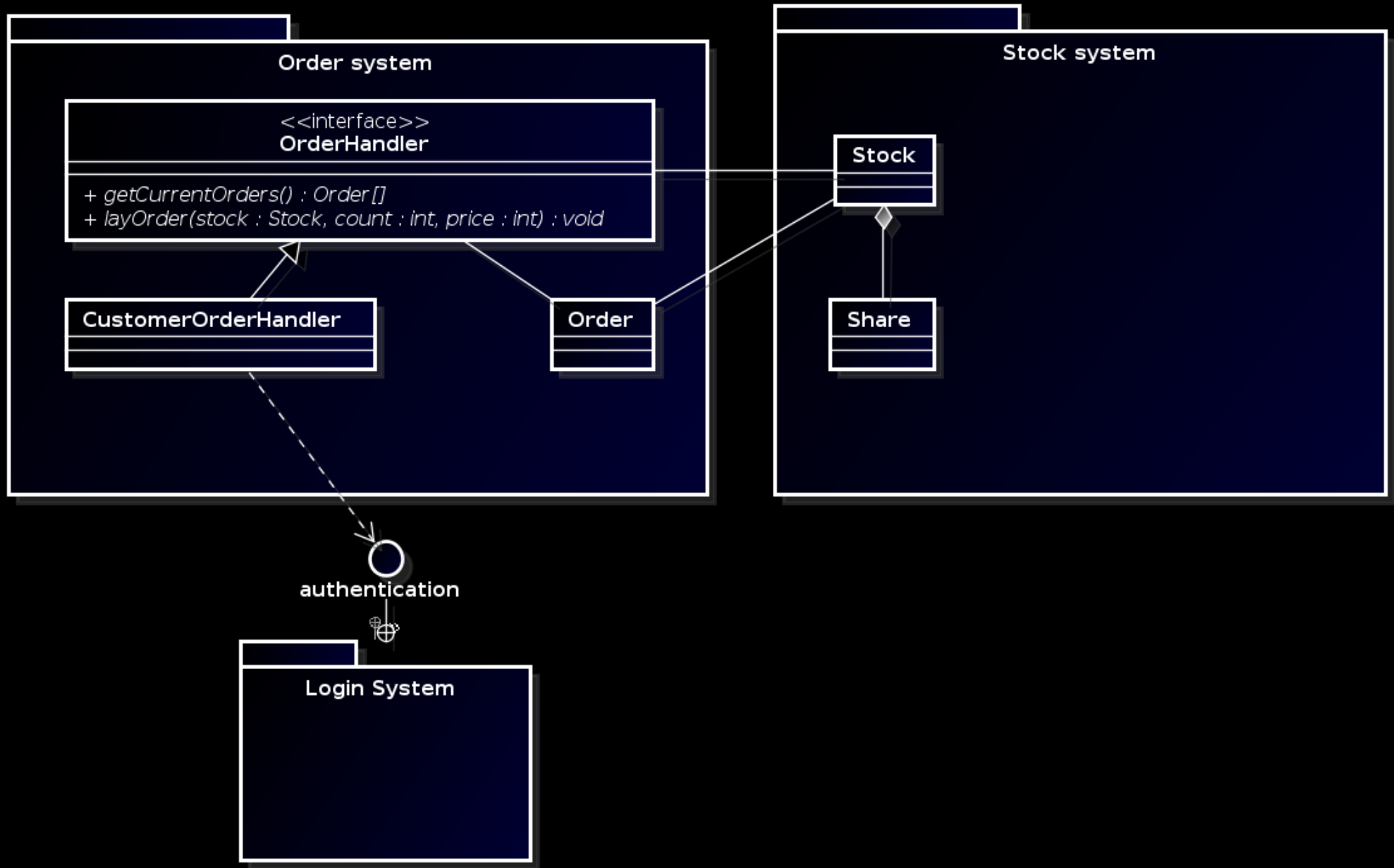
# Klassdiagram med paket



# Klassdiagram med objekt Objektdiagram

- körningsstruktur – hur det ska fungera





# Anrop mellan objekt

- Samarbetsdiagram
  - Syfte
    - Beskriva kommunikation mellan (många) objekt
  - Viktiga delar
    - objekt och deras anrop
- Sekvensdiagram
  - Syfte
    - Beskriva i detalj en komplex anropssekvens
  - Viktiga delar
    - objekt och deras livslinje
    - funktionsanrop, aktivering
    - även skapande och borttagande av objekt

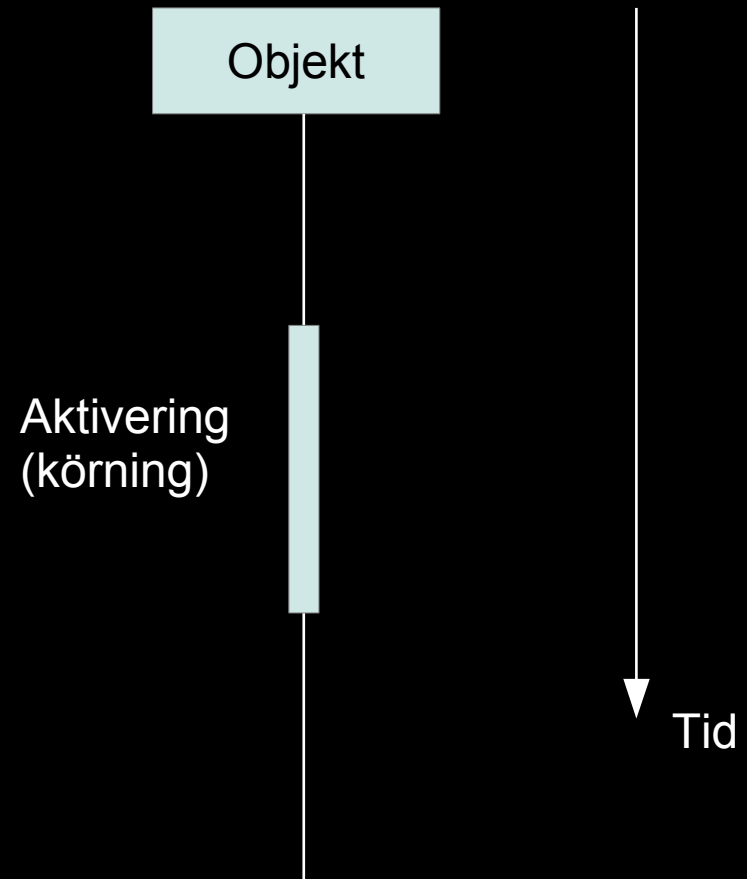


# Anrop mellan objekt

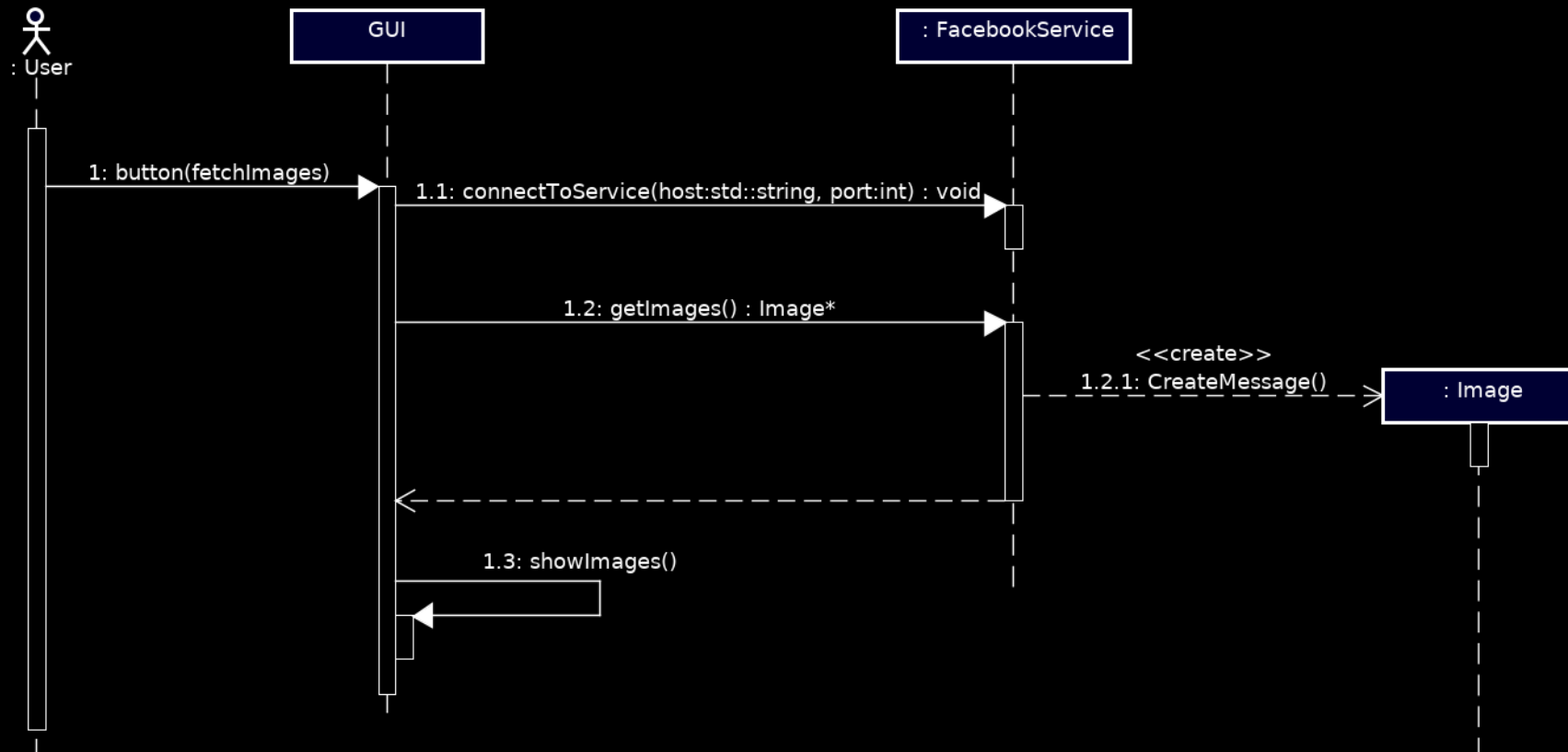
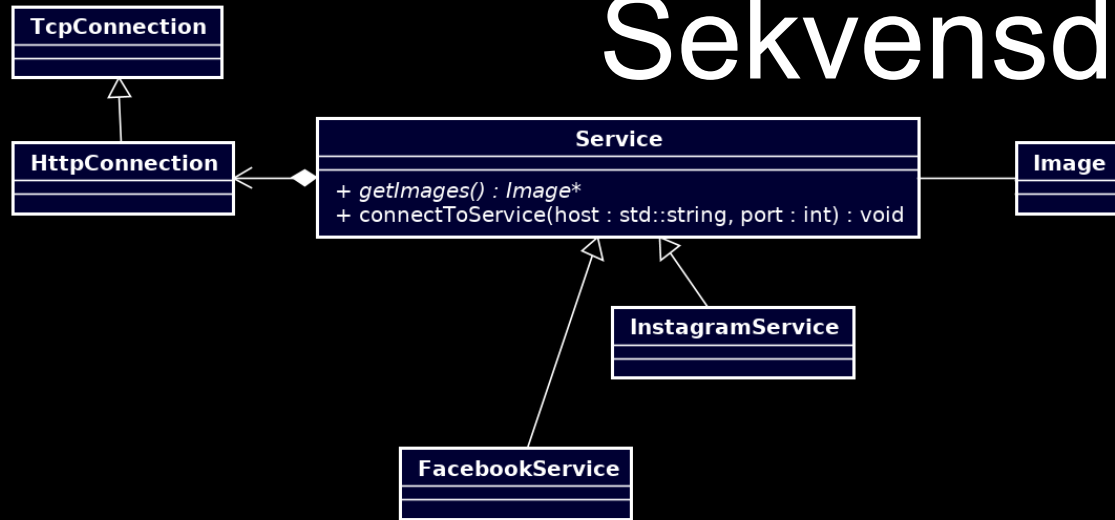
```
FacebookService service;  
service.connect("facebook.com", 80);  
  
std::vector<Image> images;  
bool success = service.getImages(images);  
  
if (!success) throw new ImageCollectionException;
```

# Sekvensdiagram

- Fokus på anropens ordning
  - tidslinje per objekt
  - aktivering per tidslinje

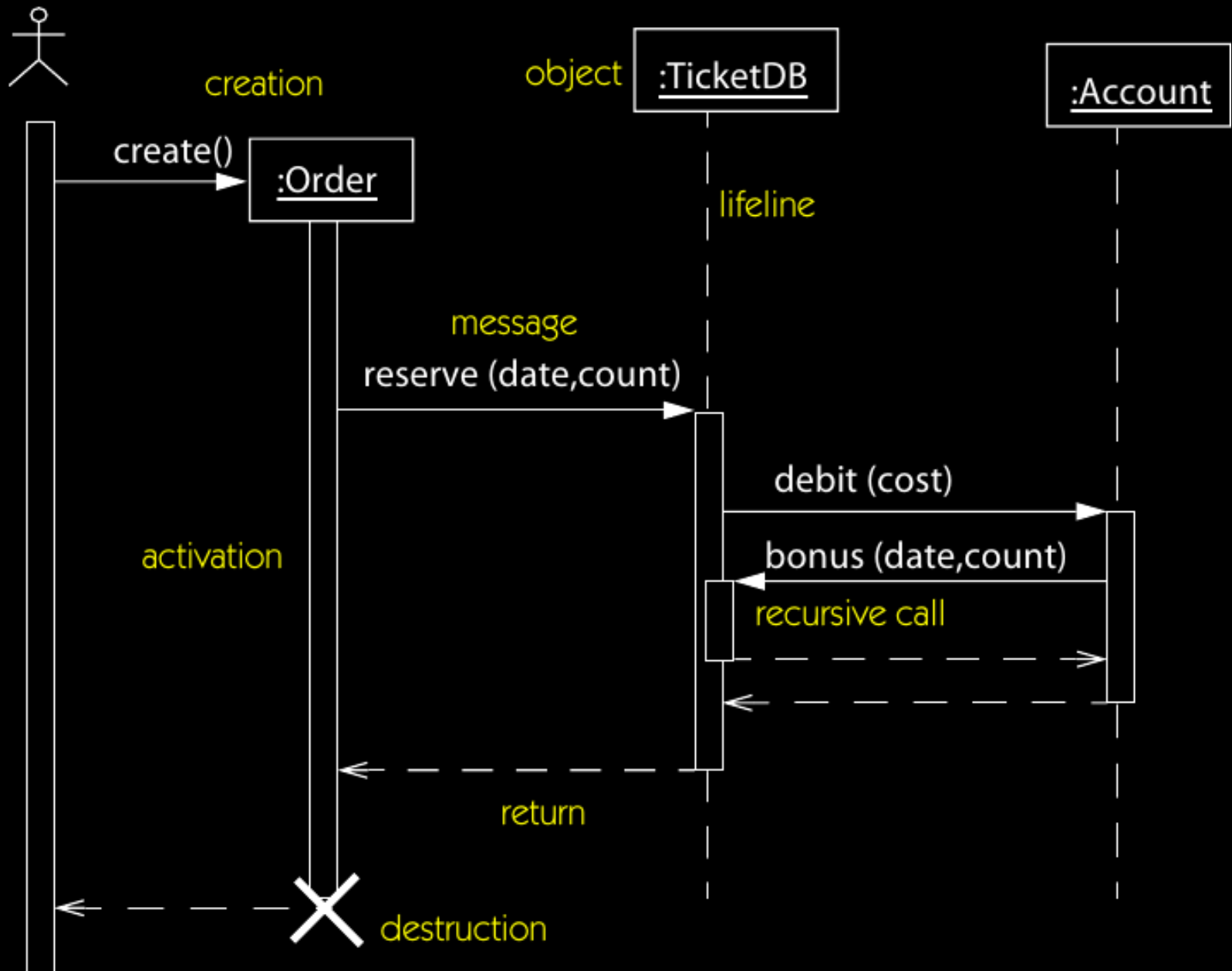


# Sekvensdiagram

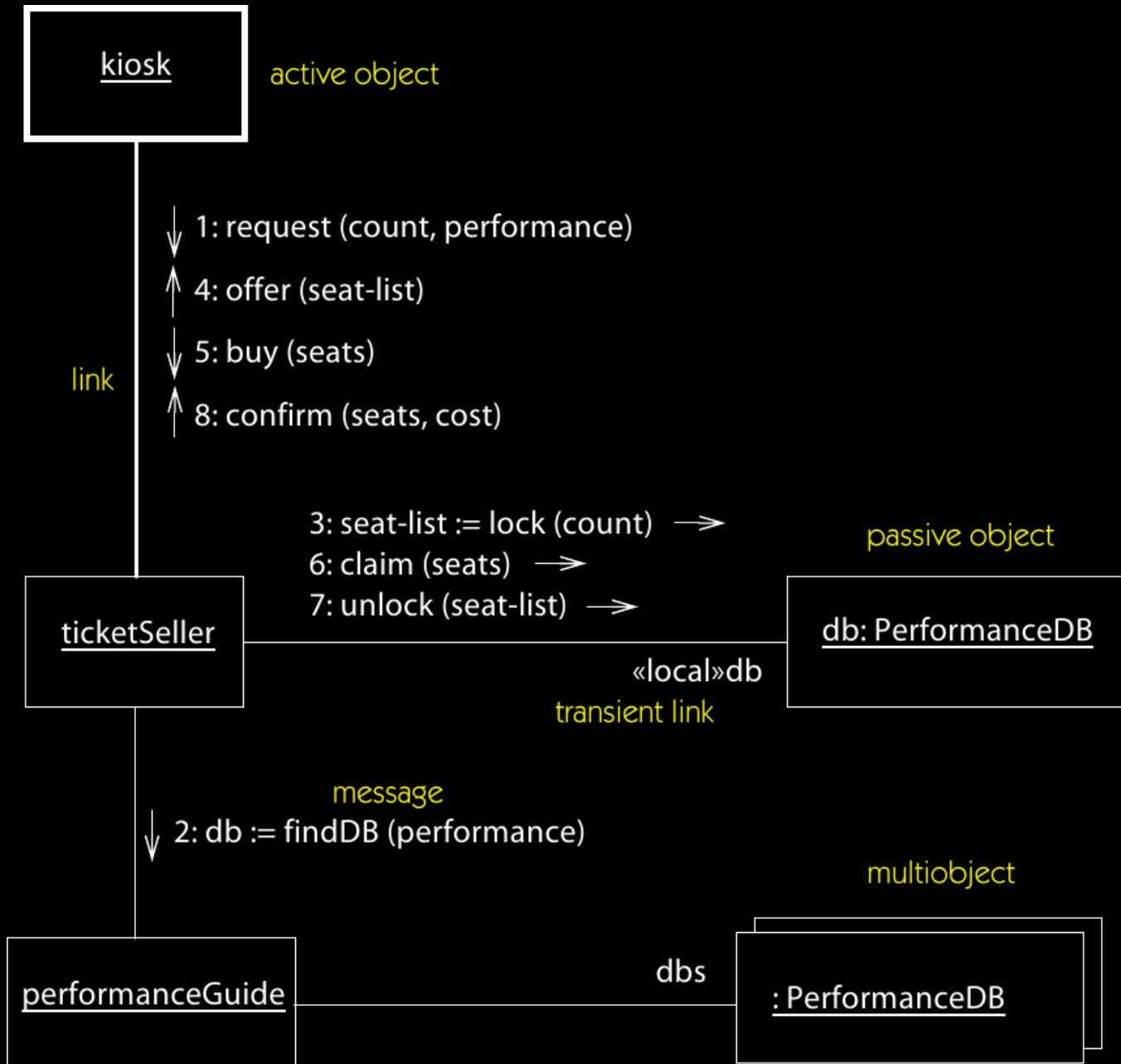


# Sekvensdiagram

an anonymous caller

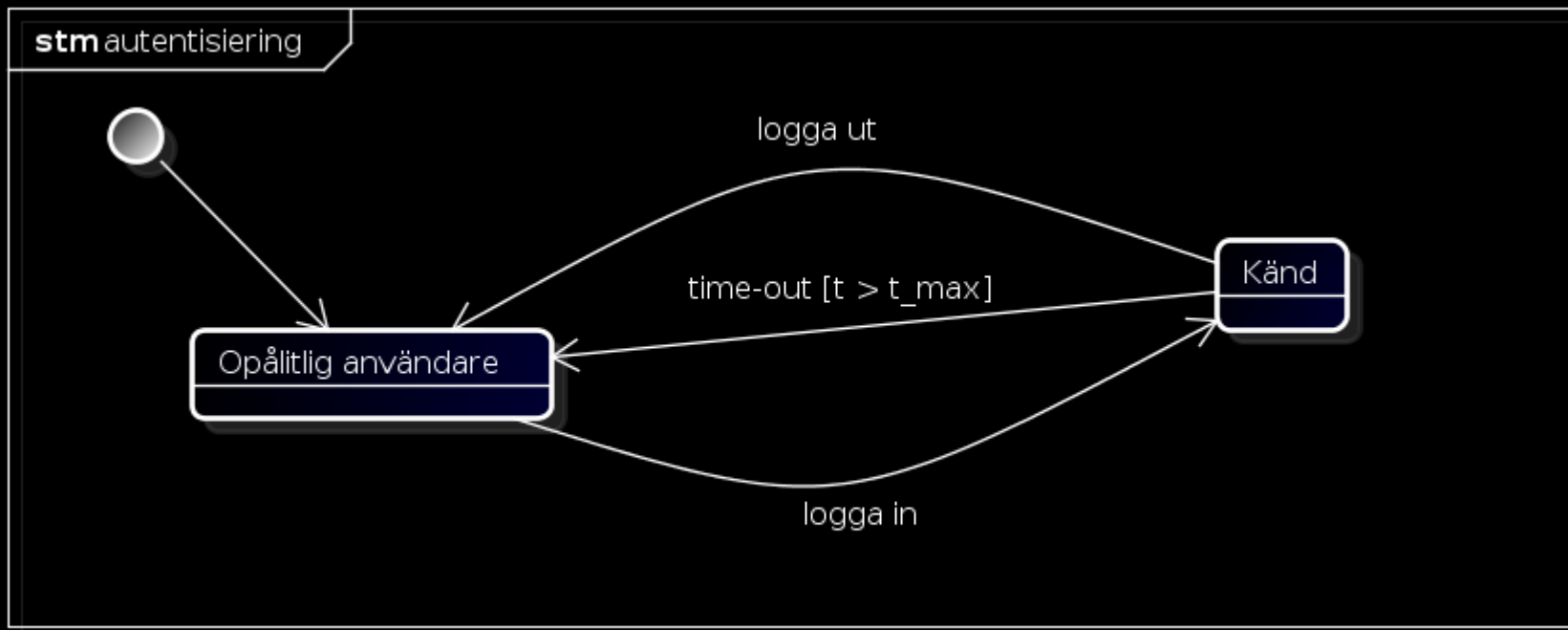


# Samarbetsdiagram

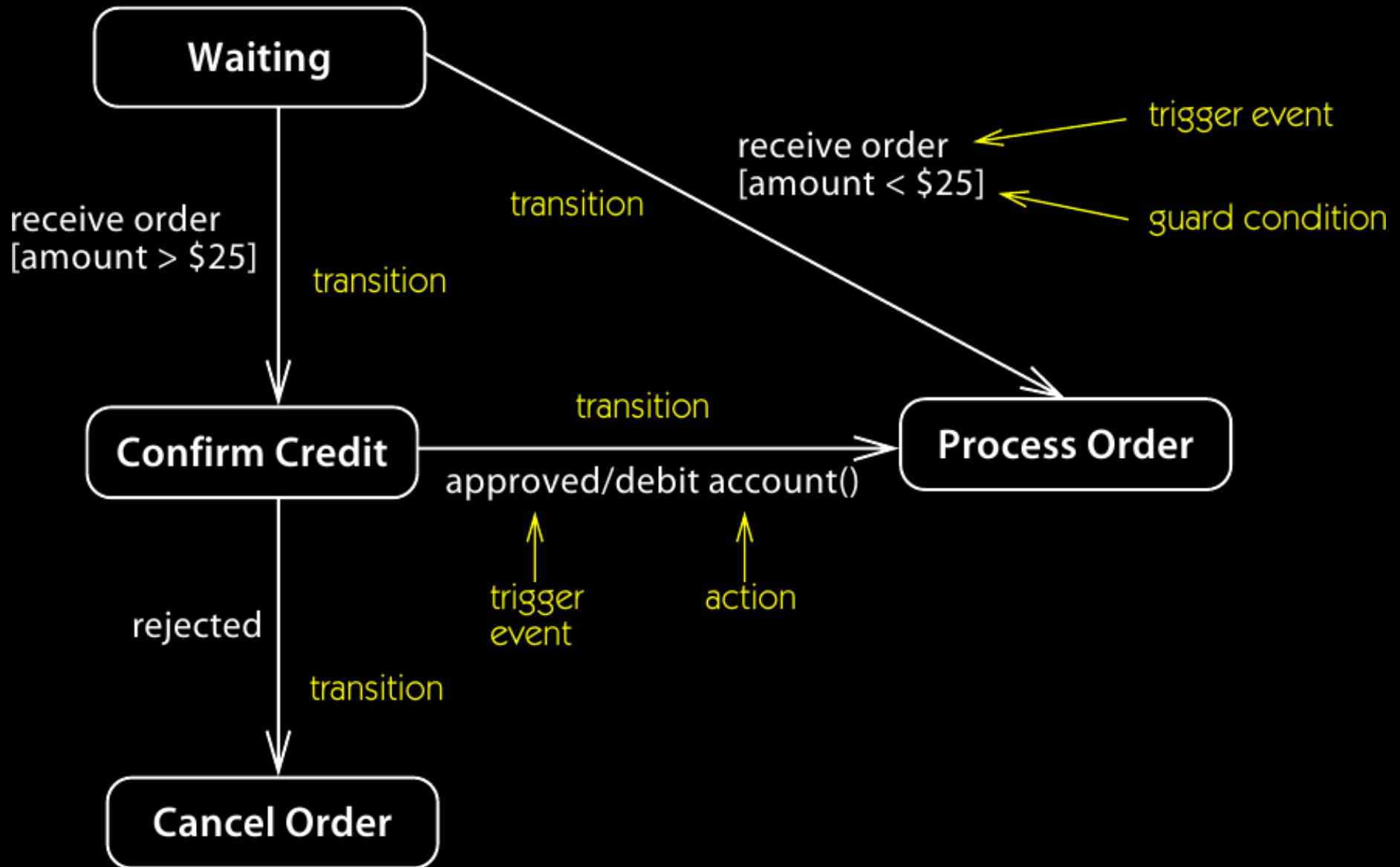


# Tillståndsdigram

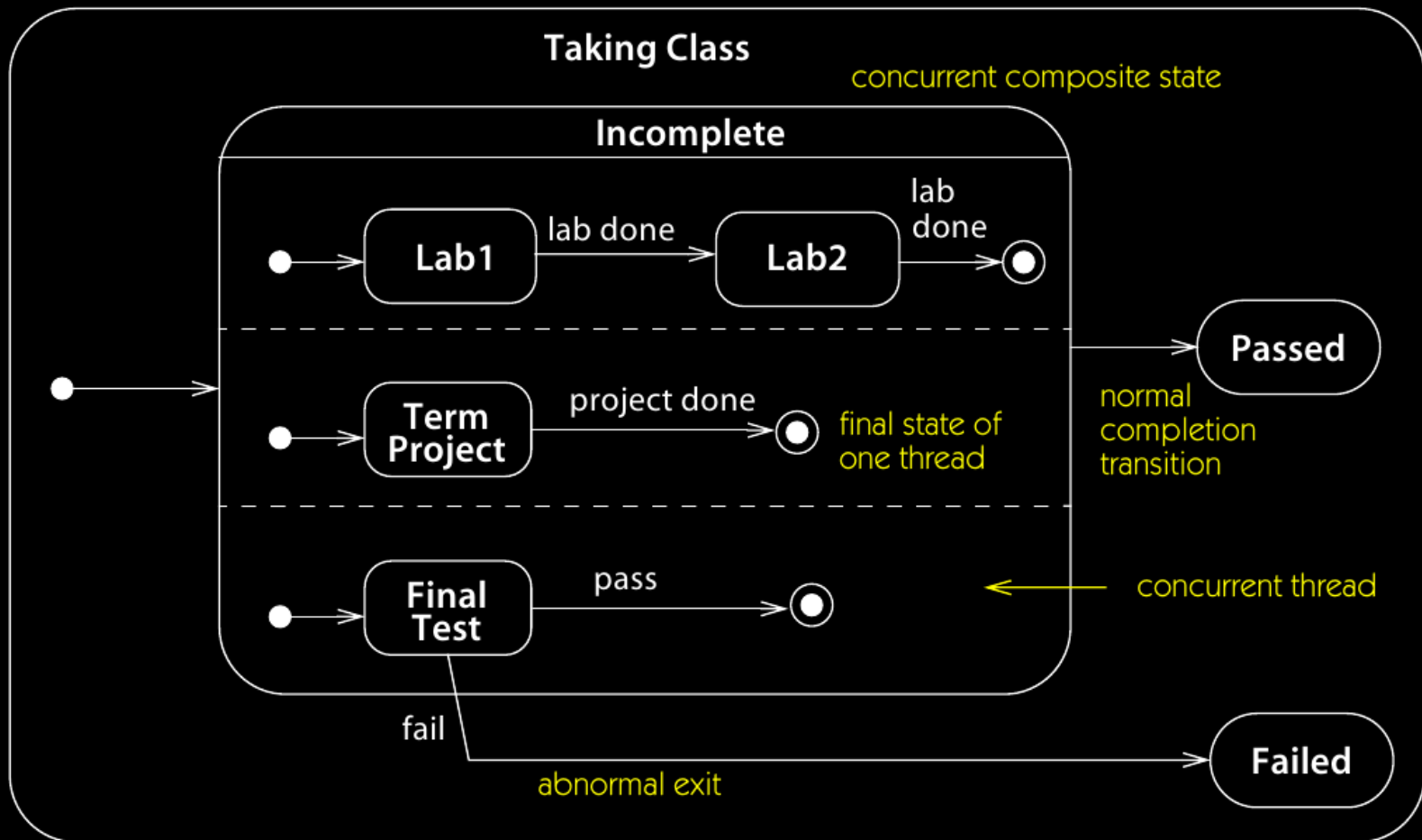
- Beskriver ett internt beteende hos en enhet
  - Hur olika händelser påverkar tillståndet hos enheten
  - Abstrakt enhet i kravspecifikationen
  - Modul i systemarkitekturen
  - Klass/objekt i programdesign



# Tillståndsdigram



# Tillståndsdigram





# UML i utvecklingsprocessen

- I stora utvecklingsprojekt
  - används för dokumentation i alla steg
- I agil utveckling
  - modellera med ett syfte – gör det som fungerar!
- Five-step UML
  - “OOAD for Short Attention Spans”
- Agile Modelling
  - “bara precis tillräcklig”-modellering genom hela projektet

# Five-step UML

- Define
  - Identifiera och strukturera systemets krav via Use case-diagram
  - Komplettera med andra diagram där det förtydligar krav och use case
- Refine
  - Beskriv scenarion för varje use case via Aktivitets-diagram
  - Komplettera med andra diagram där det tydliggör aktiviteterna
- Assign
  - Dela upp aktiviteterna i banor för att ange ansvariga moduler i systemet
- Design
  - Visa relationen mellan modulerna med Komponent-diagram
  - Komplettera med andra diagram där det tydliggör uppdelningen
- Divide and conquer
  - gå ner i nivå och utveckla komponenter och deras delar (med Klass-diagram)
  - gå upp i nivå och resonera kring hela system
  - Komplettera med andra diagram där det underlättar förståelsen för systemet

# Agil modellering

- AMDD
  - Agile Model-Driven Development
  - modeller som är “bara precis tillräckligt” för att driva utvecklingen
  - modellera med syfte – gemensam vision, förståelse, strukturering, etc
- Sprint 0 (dagar)
  - krav-modellering
  - preliminär systemarkitektur-modellering
- Iteration-modellering (timmar)
  - analys och design av sprint-backloggen
  - detaljer kring varje post
- Model storming (minuter)
  - JIT-modellering
  - grupp-diskussioner kring ett delat verktyg
  - tillsammans med kolleger och intressenter

# Modelleringsverktyg

- Computer Aided Software Engineering (CASE)
    - semi-/automatisk syntax-kontroll
    - modell-analys och kod-generator
    - kod-analys och modell-generator
    - följer standarder strikt
  - Verktyg som används i Agil utveckling
    - Papper och penna, Whiteboard, etc.
    - Använd programvara när det stödjer ditt syfte
  - Exempel
    - Astah – community-version installerad i Linux-labbet
    - UModel – integreras i Eclipse och Microsoft Visual Studio
    - Enterprise Architect
    - (PlantUML – Text-till-diagram, kan t ex integreras med Doxygen)
- Round Trip Engineering (RTE)