UML
modelovanie prípadov použitia

Use Case 7: Zadaj objednávku

Level: User-goal
Primary Actor: Zákazník

Main Success Scenario:

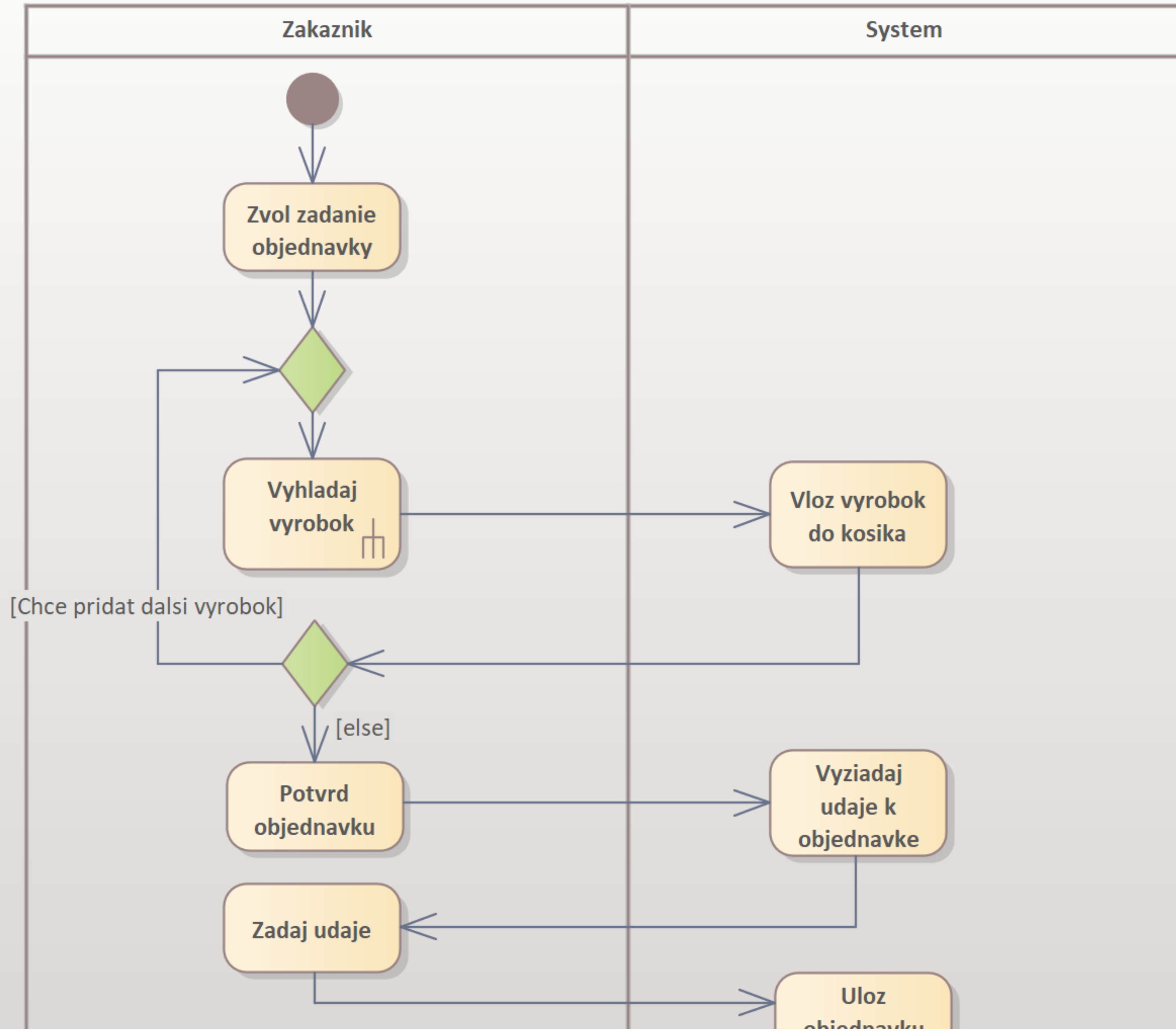
1. Zákazník zvolí zadanie objednávky.
2. Zákazník vyhľadá a zvolí výrobok (UC 35 Vyhľadaj výrobok).
3. Systém vloží zvolený výrobok do košíka.
4. Zákazník môže pokračovať vo výbere výrobkov – prípad použitia pokračuje krokom 2.
5. Zákazník objedná výrobky v košíku.
6. Systém vyžiada údaje potrebné na realizáciu objednávky vrátane spôsobu platby.
7. Zákazník zadá požadované platobné údaje.
8. Kedykoľvek počas objednávania, zákazník môže vzdať tento proces.
9. Systém uloží objednávku do zoznamu objednávok na vybavenie.
10. Prípad použitia končí.

Extensions:

9a. Stav zásob aspoň jedného výrobku z objednávky poklesol pod stanovený limit:
Systém uloží záznam do plánu doplnenia zásob o potrebe zvýšenia stavu týchto výrobkov.

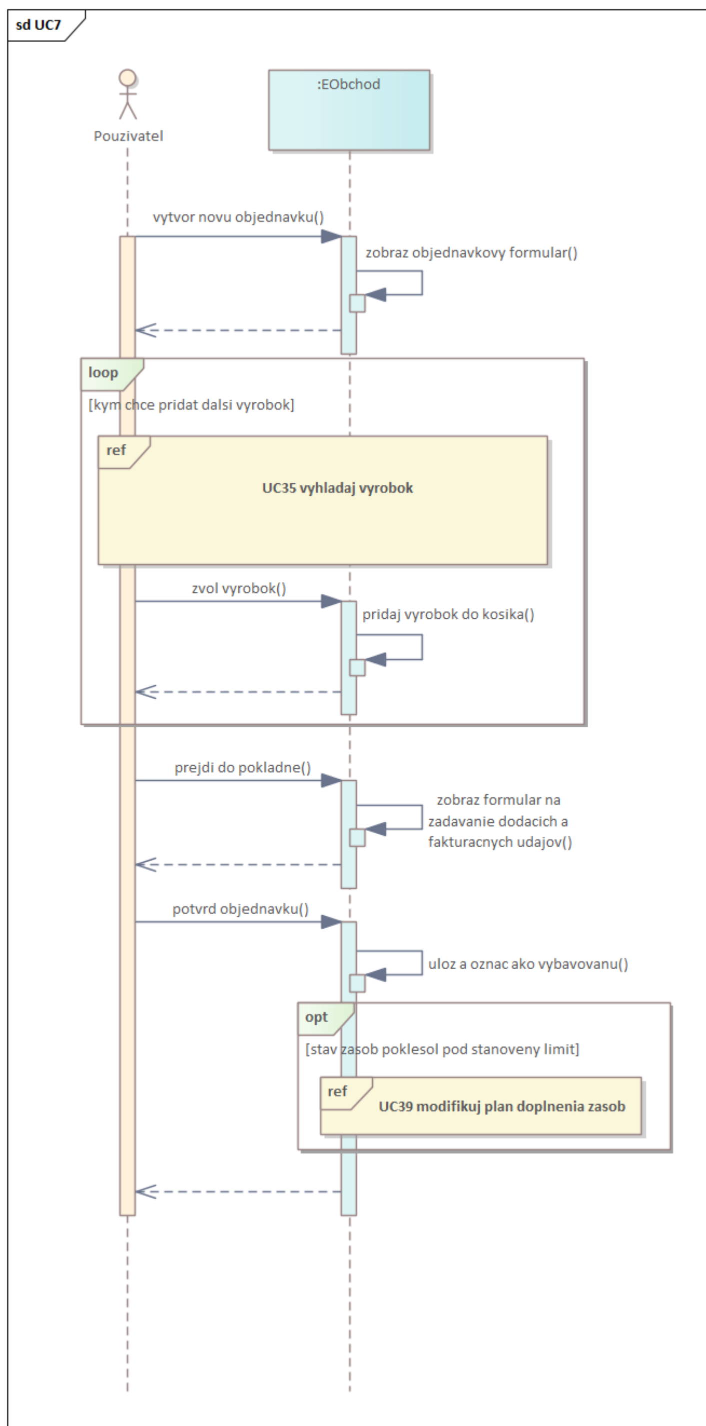
UML

modelovanie prípadov použitia
diagram aktivít



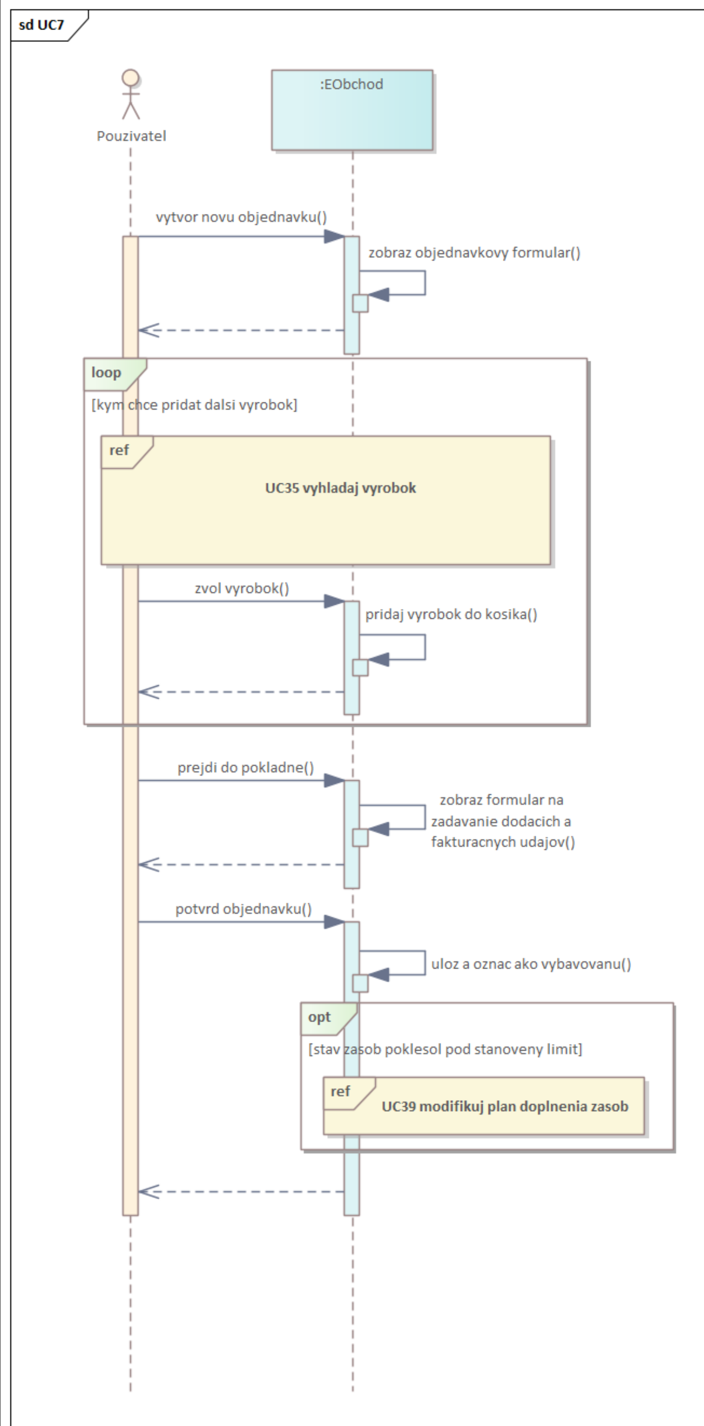
UML

modelovanie prípadov použitia sekvenčný diagram



UML

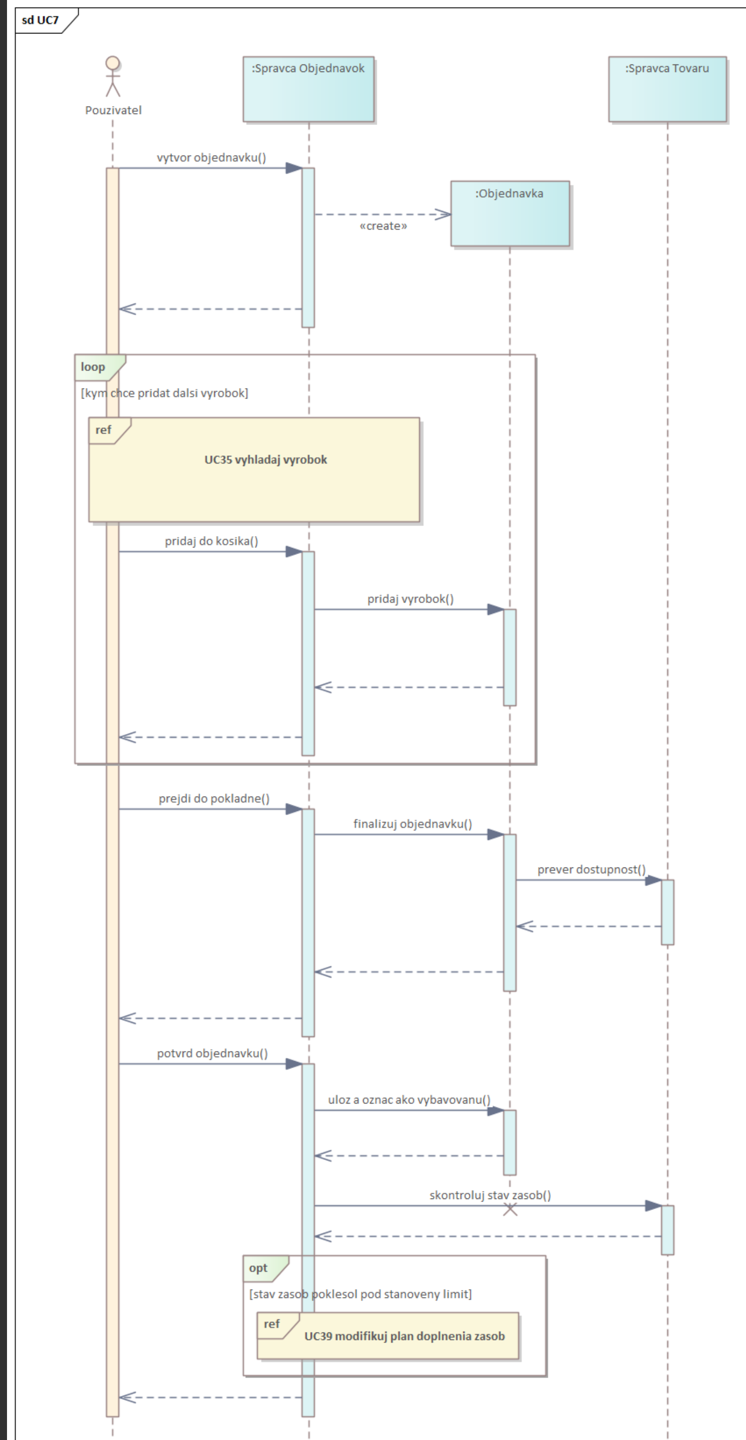
Ako transformovať analytické modely na technickú architektúru?
prechod od "Čo" k "Ako"



- Prípad použitia opisuje interakciu používateľa so systémom na vysokej úrovni abstrakcie
- Je možné vytvoriť ekvivalentný sekvenčný diagram (viď. vľavo), avšak takáto úroveň abstrakcie nebude postačujúca v ďalších fázach projektu
- Pri vyjadrení prípadu použitia diagramom sekvencií je teda žiadúce priebeh tejto interakcie postupne spresniť
- K spresneniu dochádza najmä na strane systému, v tomto prípade postupne nahrádzame inštanciu EObchod viacerými navzájom interagujúcimi inštanciami
- Týmto spresnením zavádzame nové triedy, čím v podstate začíname odvádzať štruktúru systému

UML

štruktúra z prípadov použitia

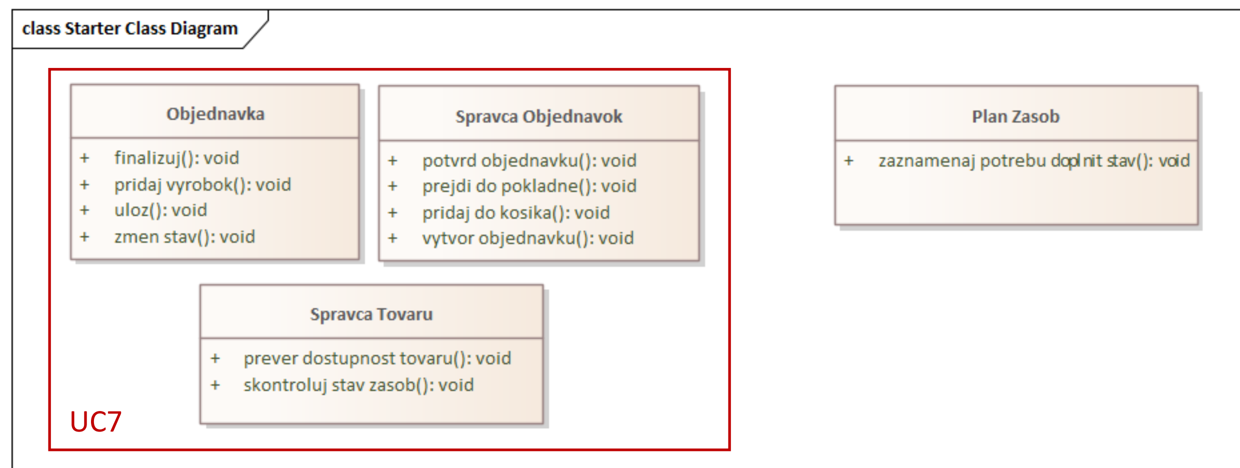


- Príklad spresnenia diagramu sekvencií
- Spresnením vznikli 3 triedy:
  - Správca Objednávok
  - Objednávka
  - Správca Tovaru

UML

štruktúra z prípadov použitia diagram tried

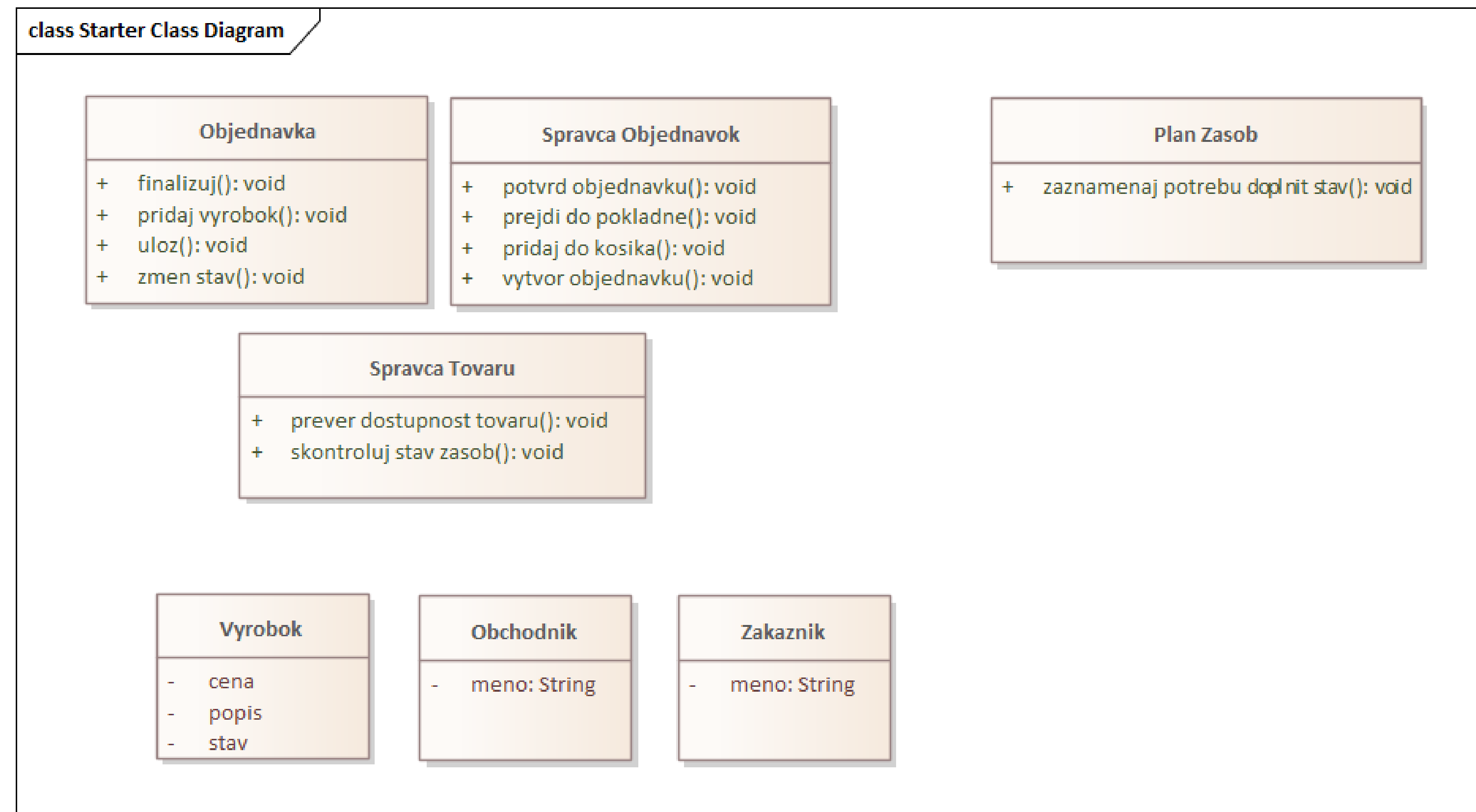
- Základ štruktúry možno získať prepisom tried, ktoré vznikli pri spresňovaní sekvenčných diagramov (vyjadrujúcich viacero prípadov použitia)



UML

štruktúra z prípadov použitia diagram tried

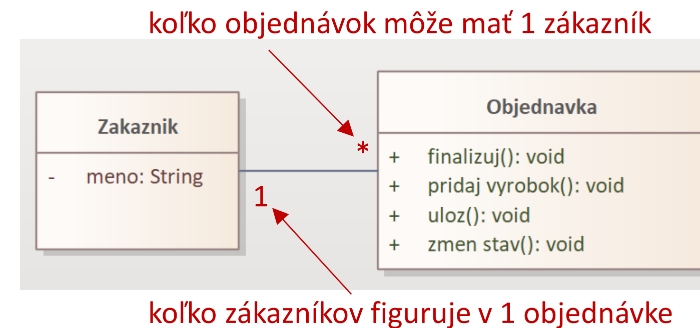
- Ďalšie triedy môžu byť identifikované na základe samotnej interakcie alebo doménových znalostí
  - Do objednávky pridávame Výrobok
  - Objednávka je kontrakt medzi Zákazníkom a Obchodníkom, pričom obe strany musia byť v rámci systému nejakým spôsobom reprezentované
  - ...



UML

štruktúra z prípadov použitia diagram tried

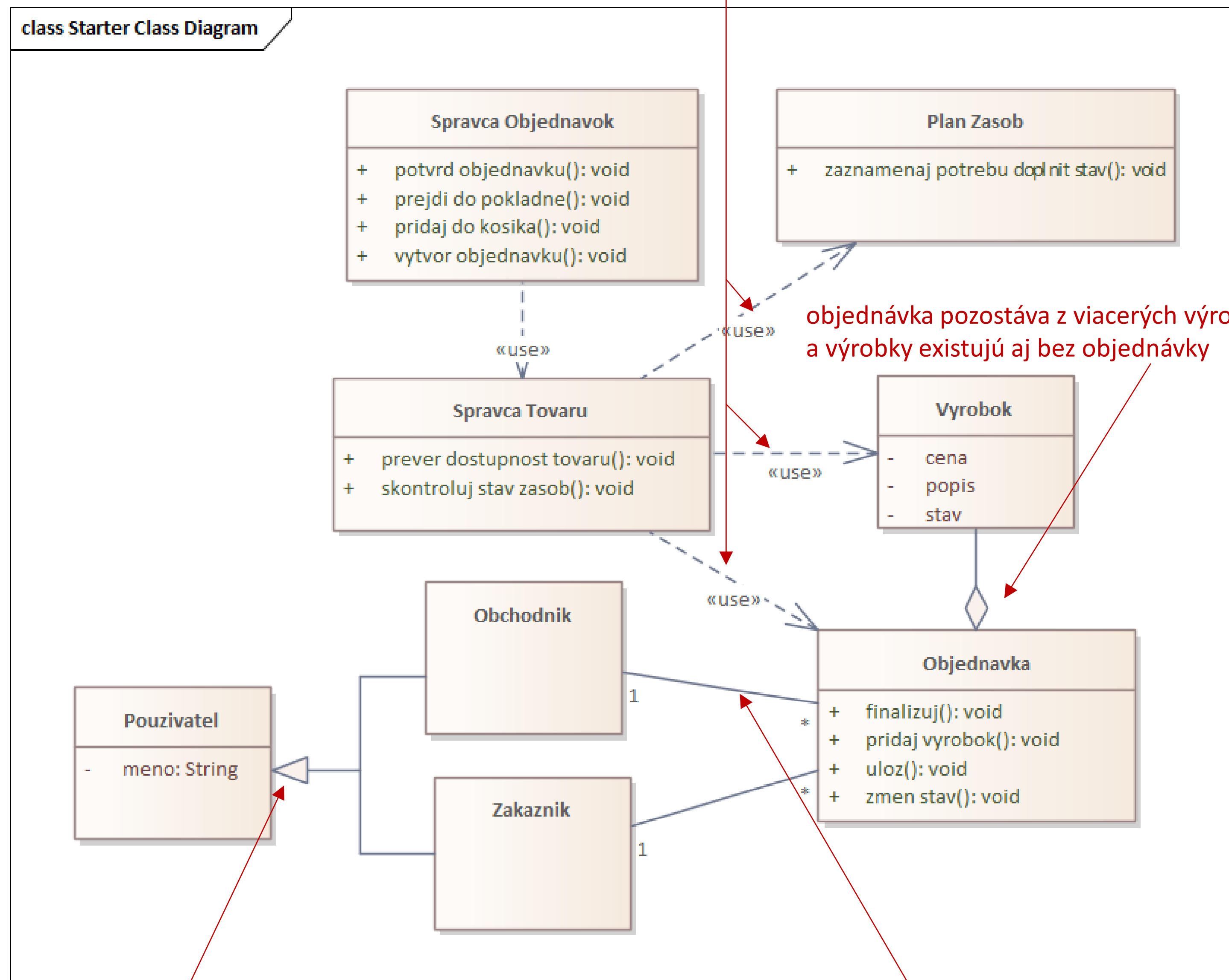
- Dôležitou súčasťou diagramu tried sú vzťahy
- Zaujíma nás hlavne:
  - Asociácia – prepojenie / súvis dvoch tried
  - Agregácia – druh asociácie, ktorý vyjadruje vzťah celok-časť medzi dvoma triedami (objednávka pozostáva z výrobkov)
  - Kompozícia – silnejší druh agregácie, kde časť je úplne závislá od celku, keď sa celok zničí, jeho časti sú tiež zničené
  - Realizácia - trieda implementuje alebo realizuje rozhranie (interface) alebo abstraktnú triedu
  - Závislosť – jedna trieda vyžaduje / potrebuje inú triedu
  - Generalizácia - "Is A" relationship, obchodník je používateľ, zákazník je používateľ, obchodník aj zákazník sú používatelia
- V rámci vzťahov vieme zachytiť aj početnosti (multiplicity)



UML

štruktúra z prípadov použitia
diagram tried

správca tovaru používa / vyžaduje plán zásob, výrobok a objednávku

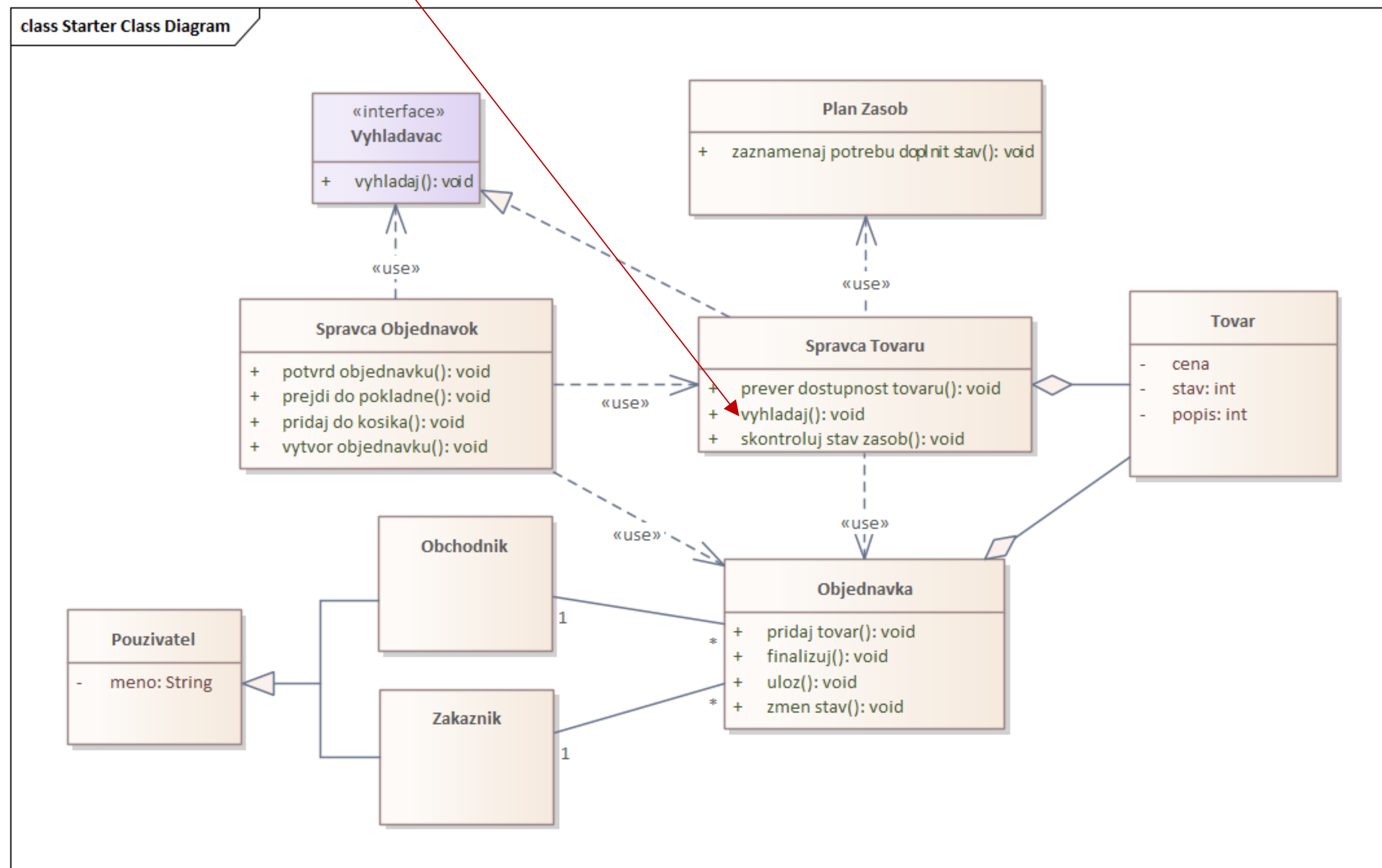


zákazník a obchodník sú používatelia
dedia vlastnosti (v tomto prípade atribút meno)

v rámci 1 objednávky sa objednáva len od 1 obchodníka
1 obchodník môže figurovať vo viacerých objednávkach

UML

štruktúra z prípadov použitia
diagram tried



UML

kód
diagram tried
realization, aggregation,
dependency

```
from abc import ABC, abstractmethod

# Vyhladavac
class SearchEngine(ABC):  # Abstract class (interface)
    @abstractmethod
    def search(self, query: str) -> list:
        """
        Perform a search based on the provided query and return
        the results.
        """
        pass


# Spravca Tovaru
class ProductManager(SearchEngine):  # Realization
    def __init__(self, products: list):
        self.products = products  # Aggregation

    def search(self, query: str) -> list:
        return [product for product in self.products if
query.lower() in product.name.lower()]

    def addNewProduct(self, orderId: int, customer: Customer,
merchant: Merchant) -> Product:
        newProduct = Product(orderId, customer, merchant) # Dep
        self.products.append(newProduct)
```

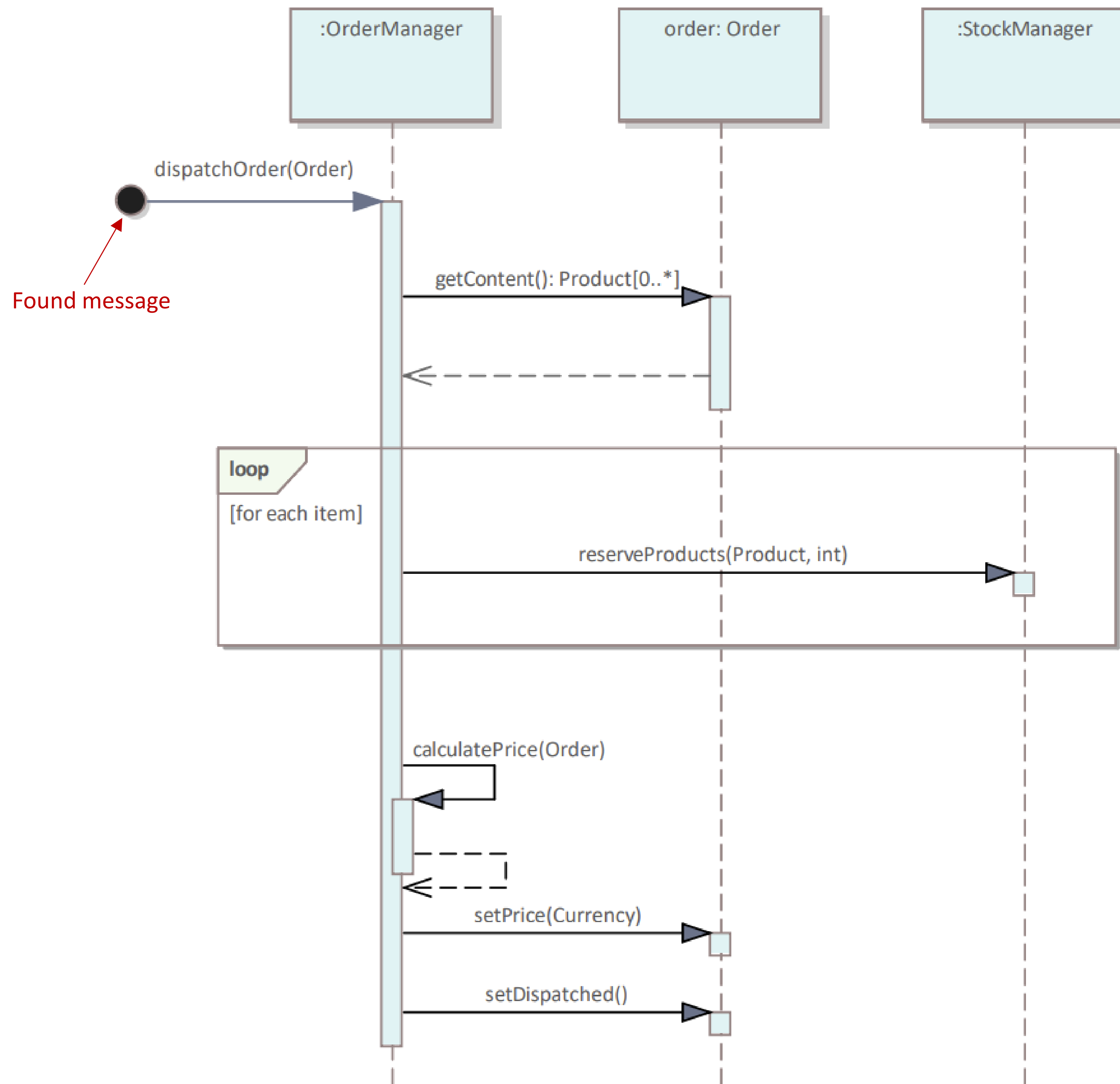
UML

kód
diagram tried
association, aggregation, generalization

```
class Product:
    def __init__(self, description: str, price: float, status: int):
        self.status = status
        self.price = price
        self.description = description


class User:
    def __init__(self, name: str):
        self.name = name

class Merchant(User): # Generalization - Merchant is a user
    pass

class Customer(User): # Generalization - Customer is a user
    pass

class Order:
    def __init__(self, orderId: int, customer: Customer, merchant: 
Merchant):
        self.orderId = orderId
        self.merchant = merchant # Association with Merchant
        self.customer = customer  # Association with Customer
        self.products = []  # Aggregation

    def addProduct(self, product) -> float:
        self.products.append(product)
```

UML

Detailný model operácie
Sekvenčný diagram



Štýly a vzory

Ako zachovať poriadok a vnášať overené riešenia?

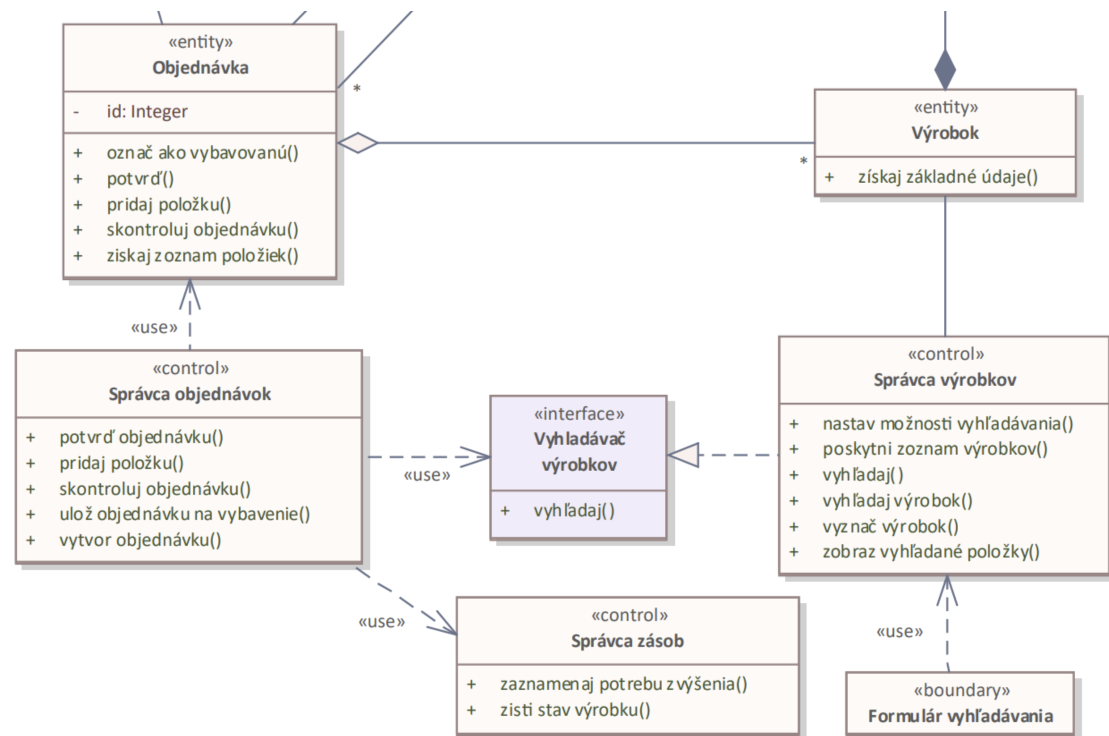
- Štýly vs vzory
  - https://www.geeksforgeeks.org/types-of-software-architecture-patterns/#software-architecture-pattern-vs-design-pattern
- Vzory
  - https://refactoring.guru/design-patterns

Features	Software Architecture Pattern	Design Pattern
Definition	This is a high-level structure of the entire system.	This is a low-level solutions for common software design problems within components.
Scope	Broad, covers entire system.	Narrow, focuses on individual components.
Purpose	Establish entire system layout.	Provide reusable solutions for the recurring problems within a systems' implementation.
Focus	System stability, structural organization.	Behavioral and structural aspects within components.
Documentaion	It involves architectural diagrams and high-level design documents.	It includes UML diagrams, detailed design specifications.
Examples	Layered Architecture, Microservices, Client-Server.	Singleton, Factory, Strategy, Observer.

UML

štruktúra z prípadov použitia
alternatívny spôsob
klasifikácia tried ako pomôcka

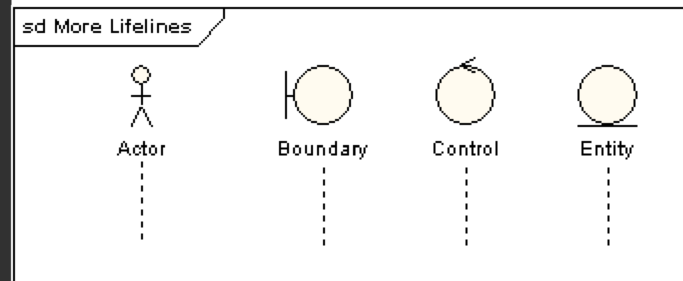
- There are several nonstandard but routinely used class stereotypes available in several UML tools including IBM Rational Software Architect (RSA) and Sparx Enterprise Architect: «Boundary», «Control», «Entity».
    - *entity* – triedy prevažne údajového charakteru, v ktorých sú kľúčové atribúty
    - *control* – triedy prevažne procesného charakteru, v ktorých sú kľúčové operácie
    - *boundary* – triedy rozhrania voči používateľovi alebo iným systémom
- Pozor na sekvenčný diagram (stále treba uviesť typ)

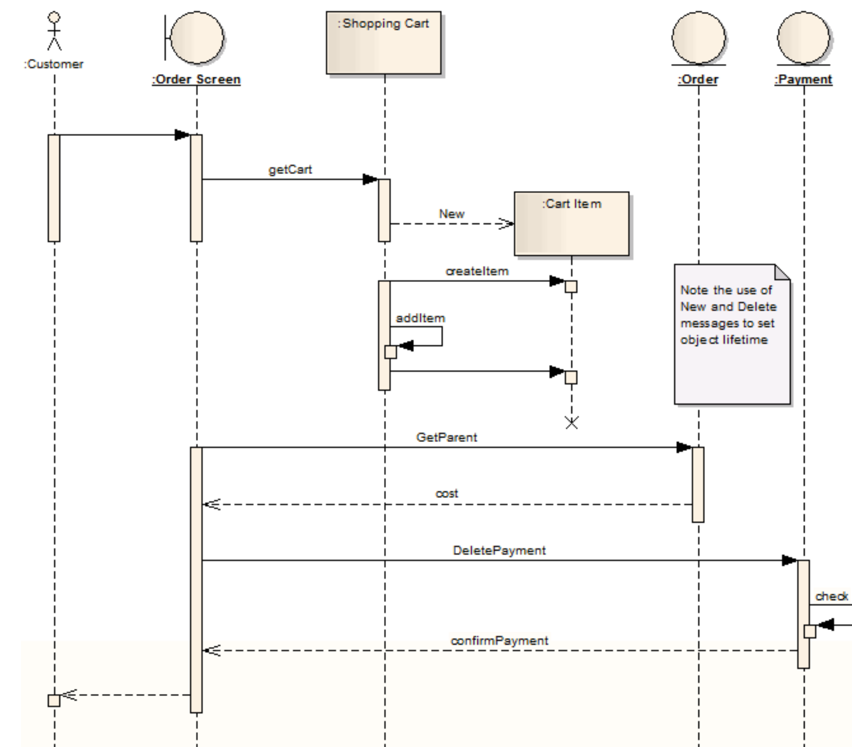


UML

sekvenčný diagram klasifikácia tried

- There are several nonstandard but routinely used class stereotypes available in several UML tools including IBM Rational Software Architect (RSA) and Sparx Enterprise Architect: «Boundary», «Control», «Entity».
  - *entity* – triedy prevažne údajového charakteru, v ktorých sú kľúčové atribúty
  - *control* – triedy prevažne procesného charakteru, v ktorých sú kľúčové operácie
  - *boundary* – triedy rozhrania voči používateľovi alebo iným systémom





Unified Process

(malá metodická vsuvka kvôli BCE)

- BCE klasifikácia pochádza z metodiky Unified Process
- Snaha zabezpečiť, aby bola architektúra odolná voči zmenám

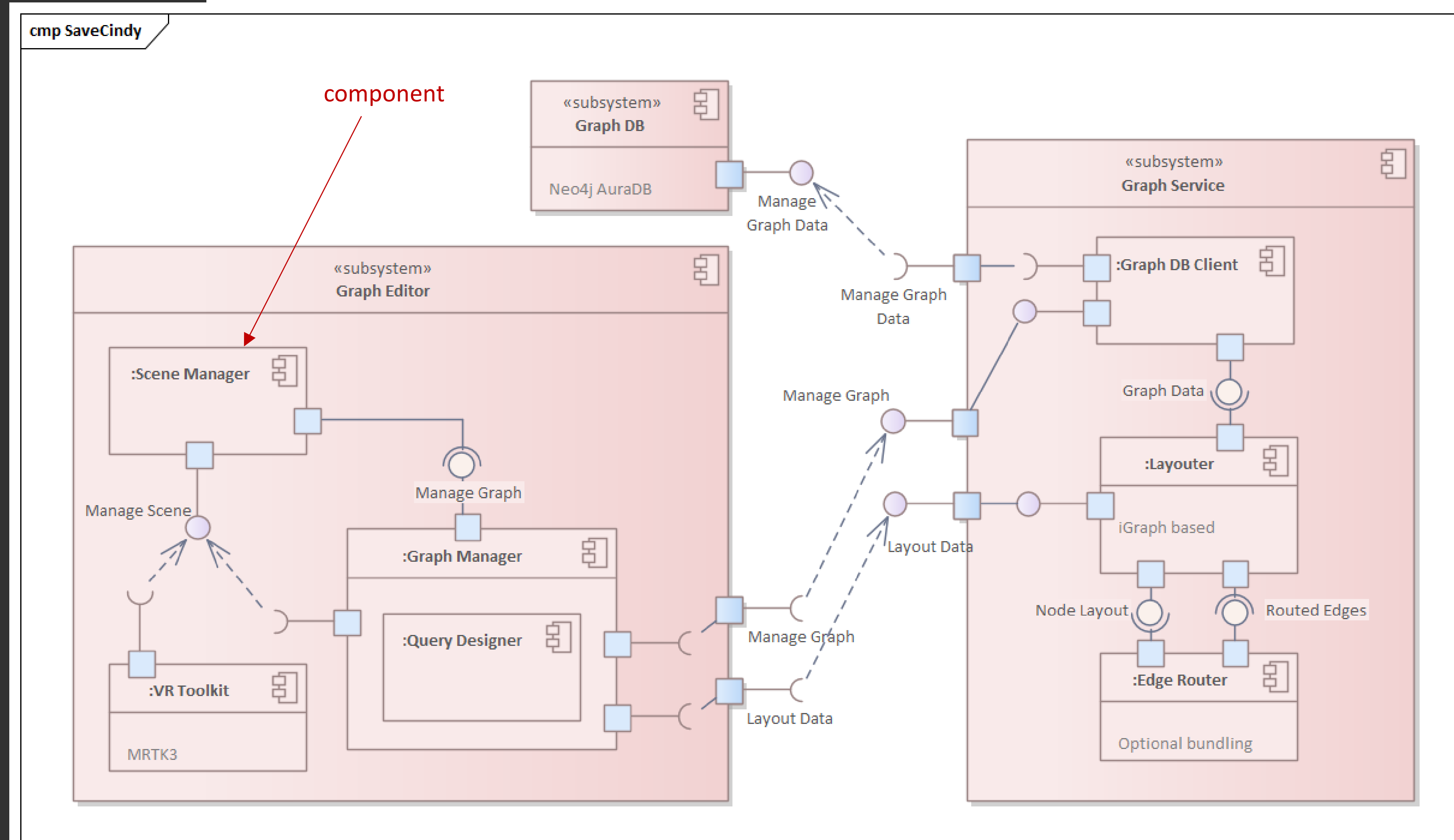
Čo je Unified Process?

- Rámec tvorby softvéru
- Používa sa pri definovaní činností a ich postupností v rámci softvérového projektu
- Definuje 4 fázy, ktorými každý projekt prejde:
  - Inception - definovanie vízie a rozsahu projektu
  - **Elaboration** - zvládnutie rizík a návrh stabilnej architektúry
  - Construction - vybudovanie kompletného systému
  - Transition - nasadenie systému u používateľa
- V rámci každej fázy prebehne jedna alebo viac iterácií cez tzv. disciplíny (disciplines), ktoré zhruba zodpovedajú etapám životného cyklu softvéru
- Ide o iteratívno-inkrementálny rámec zameraný na **vývoj vedený prípadmi použitia**

UML

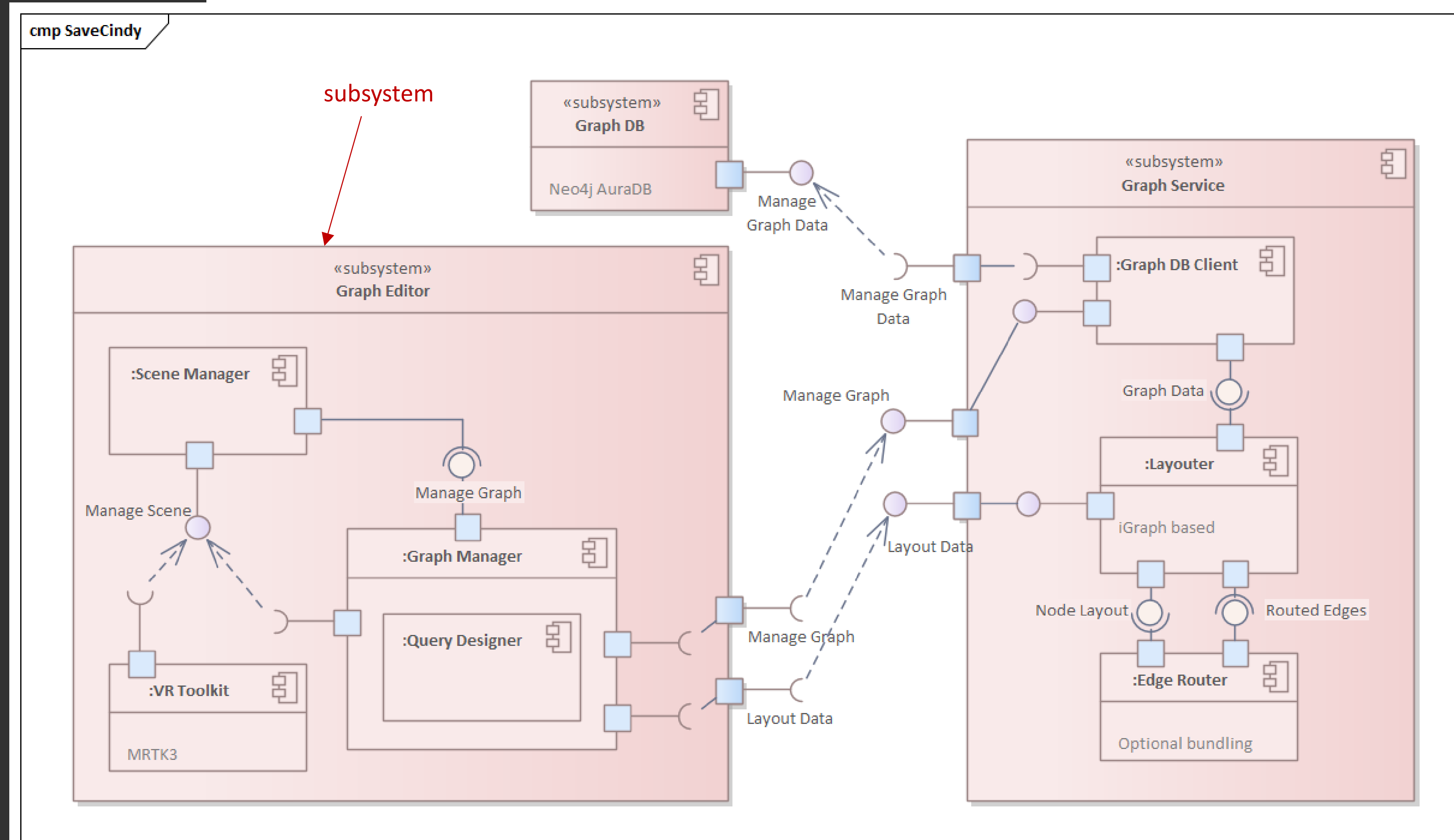
Z akých hlavných častí bude systém pozostávať?
diagram komponentov

- **Component** - replaceable part of a system that encapsulates its functionality and exposes its services through well-defined interfaces



UML
diagram komponentov

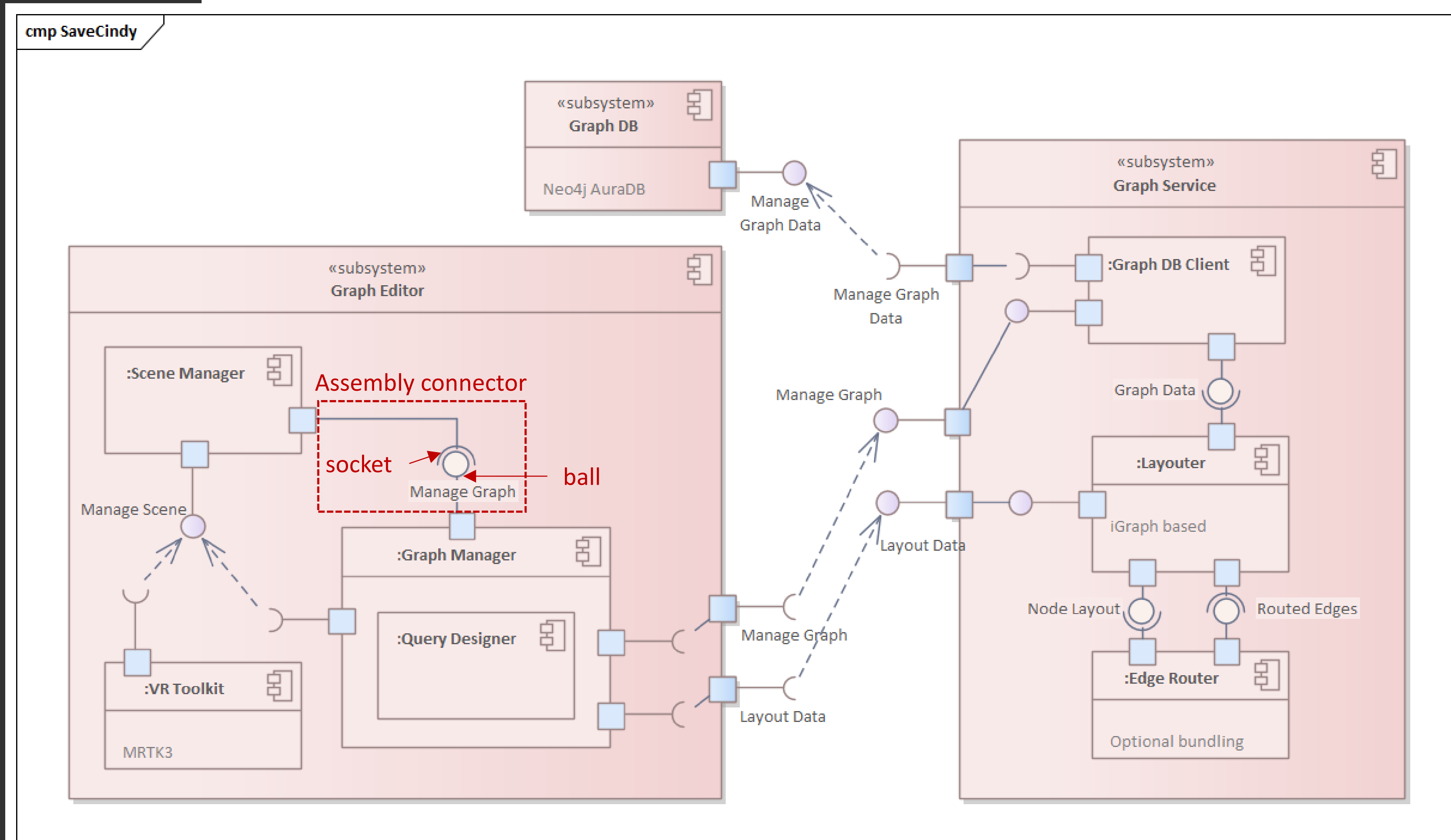
- **Subsystem**
  - A stereotyped component that represents independent, behavioral unit in a system.
  - A unit of hierarchical decomposition for large systems. A subsystem is commonly instantiated indirectly.



UML

diagram komponentov

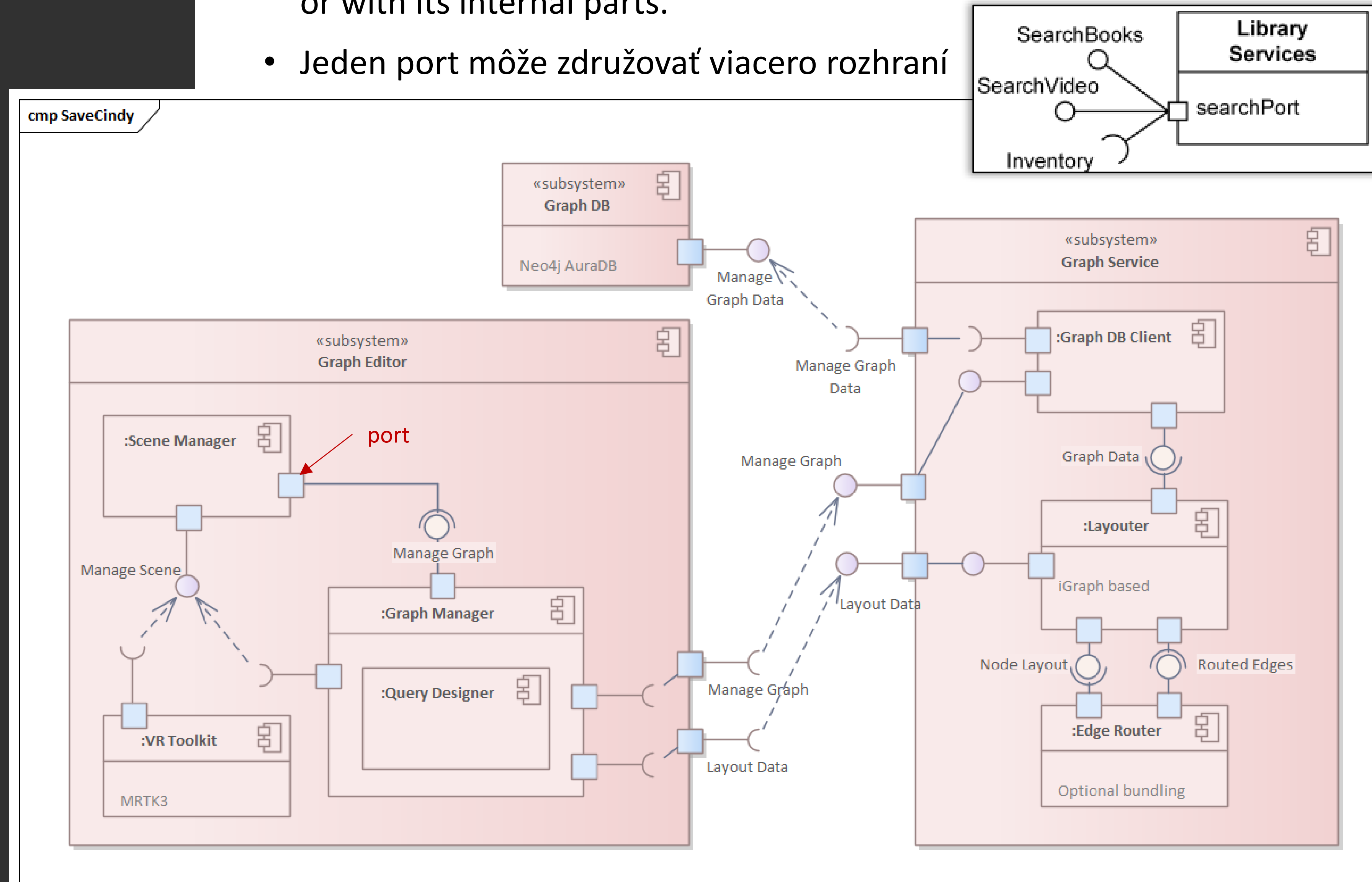
- **Assembly connector** - *a connector* between two or more parts or ports on parts *that defines that one or more parts provide the services that other parts use.*
- Používame ball-and-socket notáciu



UML

diagram komponentov

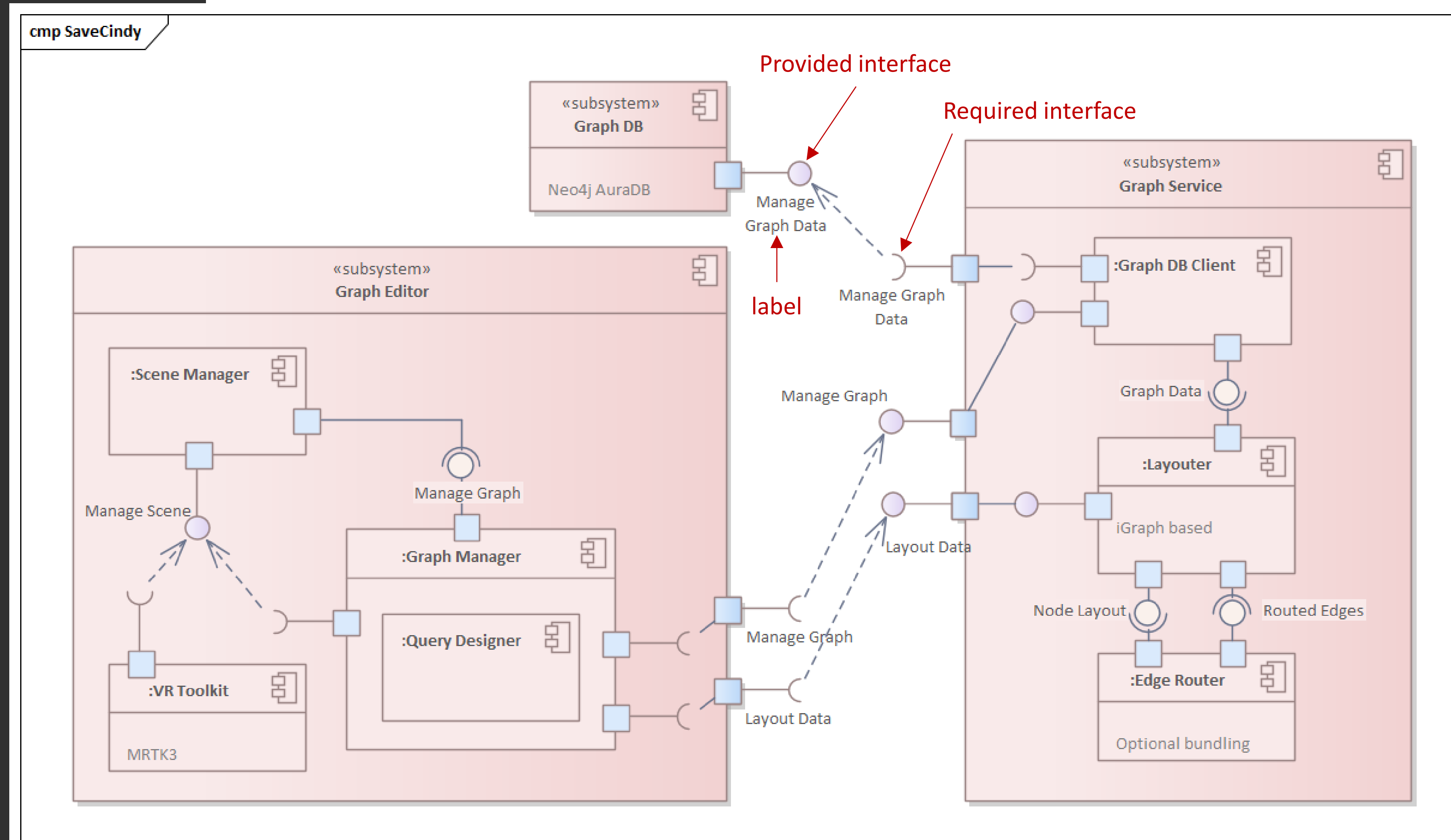
- **Port -** specifies an interaction point through which a classifier can communicate with its environment, with other classifiers, or with its internal parts.
- Jeden port môže združovať viacero rozhraní



UML

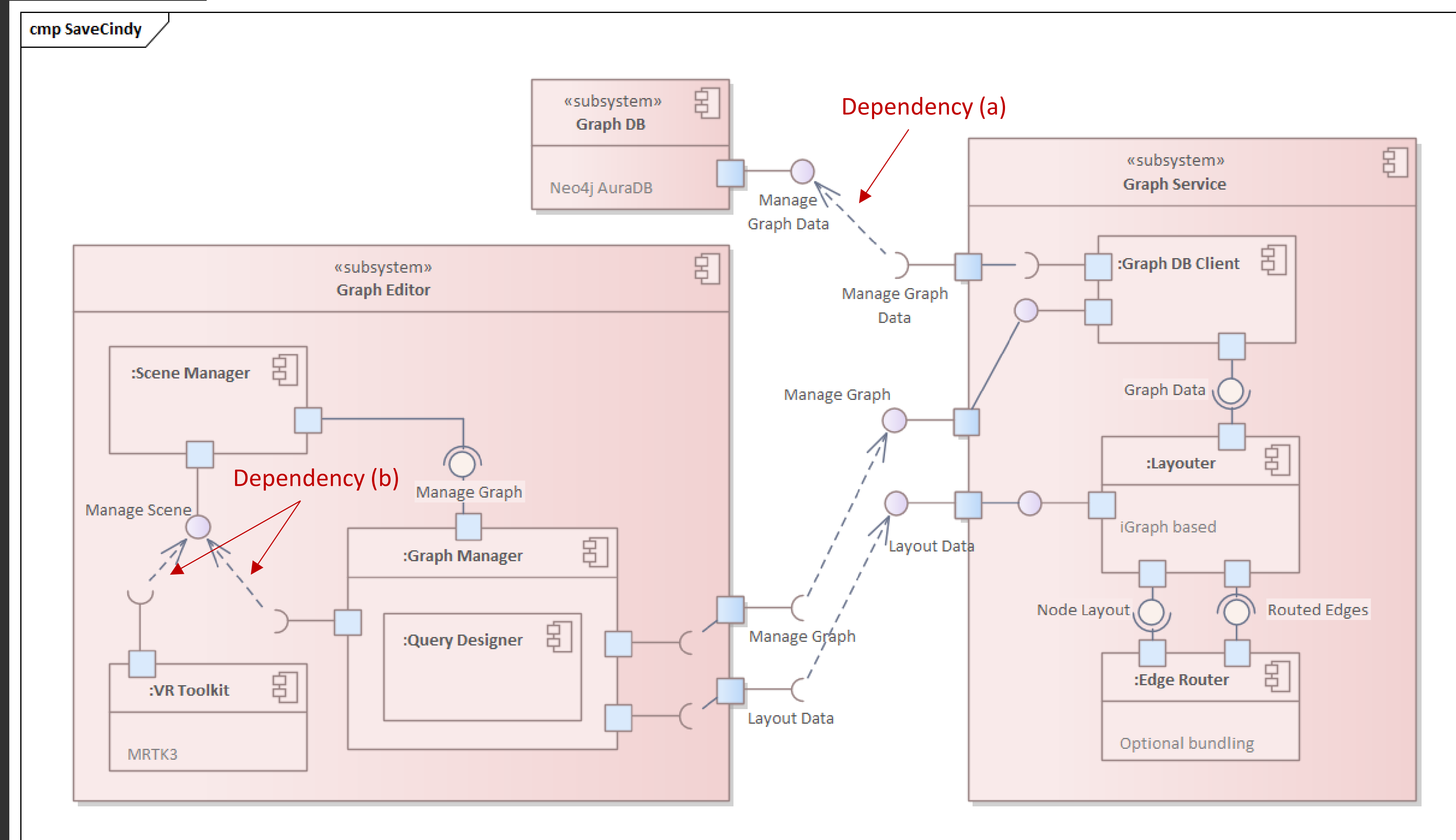
diagram komponentov

- **Provided interface** – čo komponent ponúka? Aké rozhranie / službu / funkcionalitu?
- **Required interface** – čo komponent naopak vyžaduje / potrebuje?
- Je nutné pomenovať (viď. label)!



UML
diagram komponentov

- **Dependency** budeme používať pri prepájaní provided a required interfaces v dvoch prípadoch
  a) Ak komponenty nie sú inštancie (napr. subsystem), keďže assembly možno používať podľa špecifikácie len na prepájanie inštancií
  b) Ak chceme na jeden provided interface napojiť viacero required interfaces (EA nepodporuje iný spôsob)

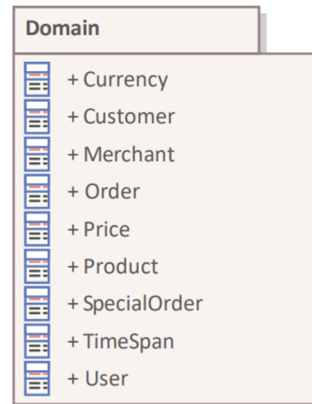


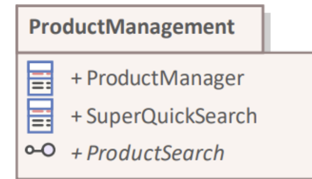
UML

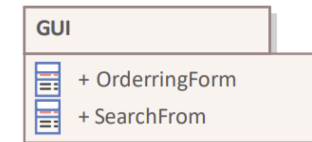
Ako sú triedy logicky organizované a aké sú medzi nimi hranice zodpovednosti?

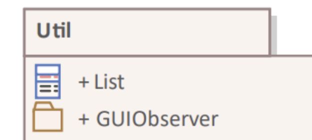
diagram balíkov

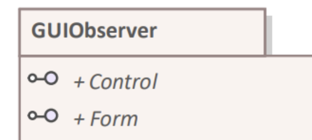
základné elementy







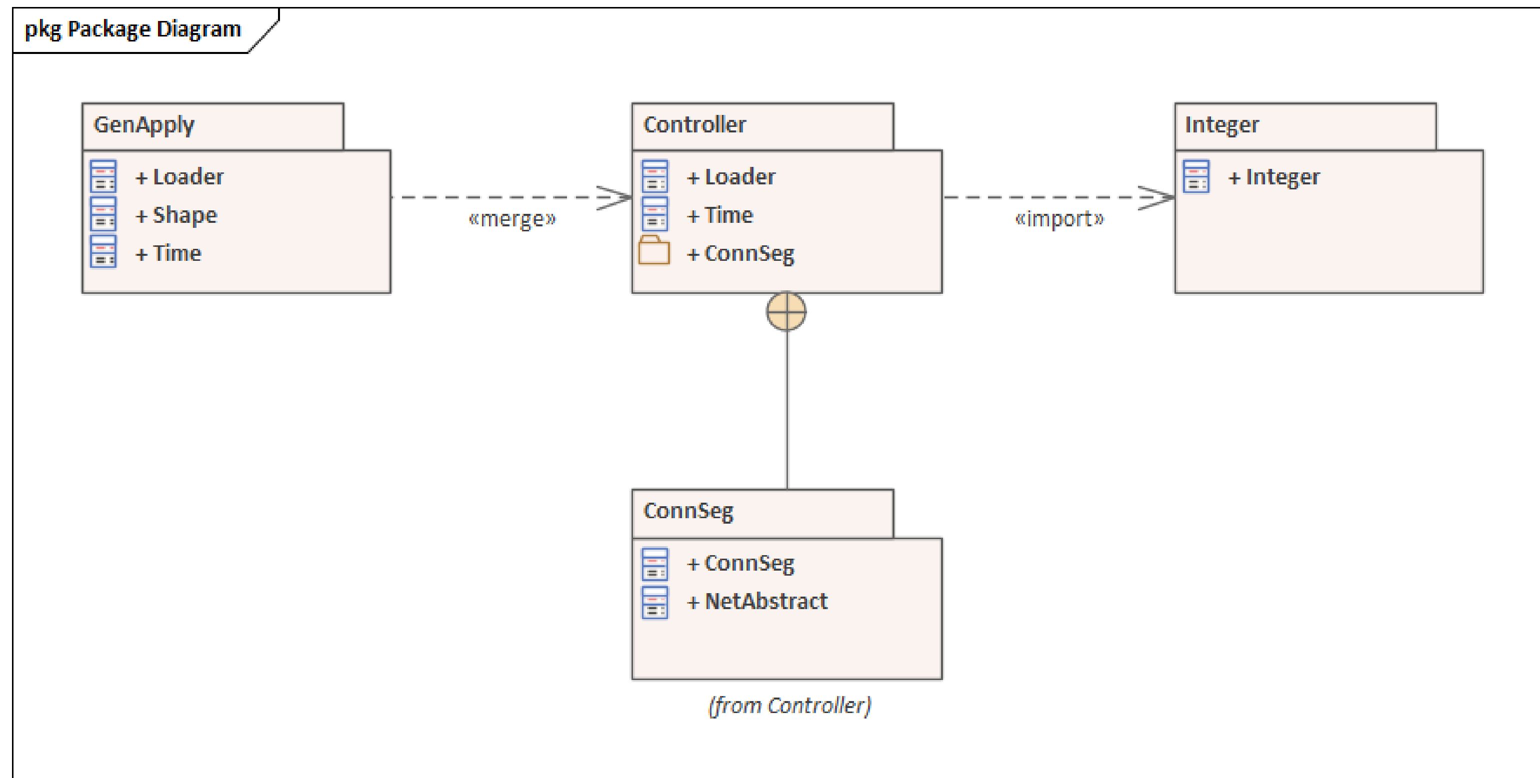




UML

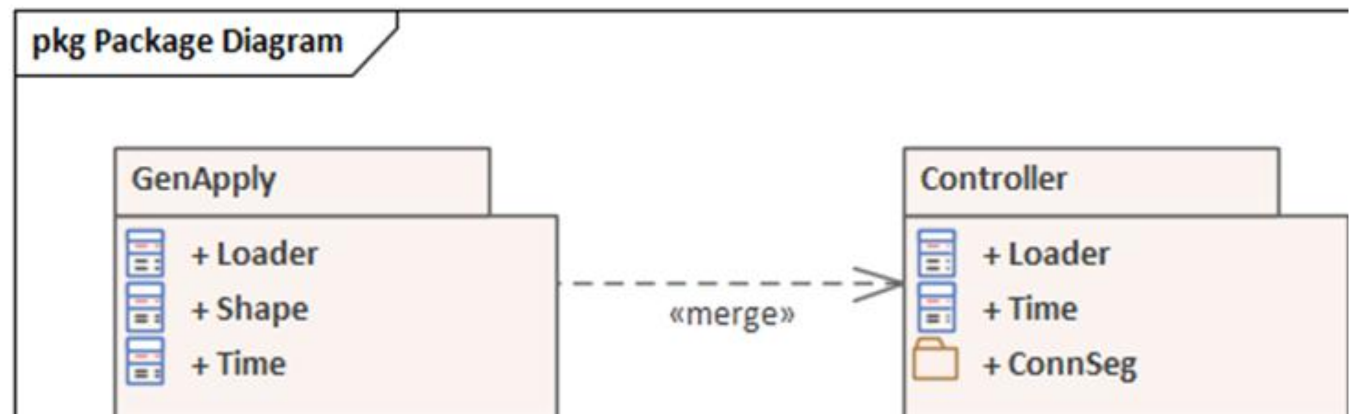
diagram balíkov
vzťahy

- Budeme používať 3 základné vzťahy
  - Nesting – „The Nesting Connector is an alternative graphical notation for expressing containment or nesting of elements within other elements. It is most appropriately used for displaying Package nesting in a Package diagram. "
  - Merge – „A PackageMerge is a directed relationship between two Packages that indicates that the contents of the target mergedPackage are combined into the source receivingPackage" (like generalization).
  - Import – „A Package Import relationship is drawn from a source Package to a Package whose contents have been imported. "



UML
diagram balíkov vzťahy

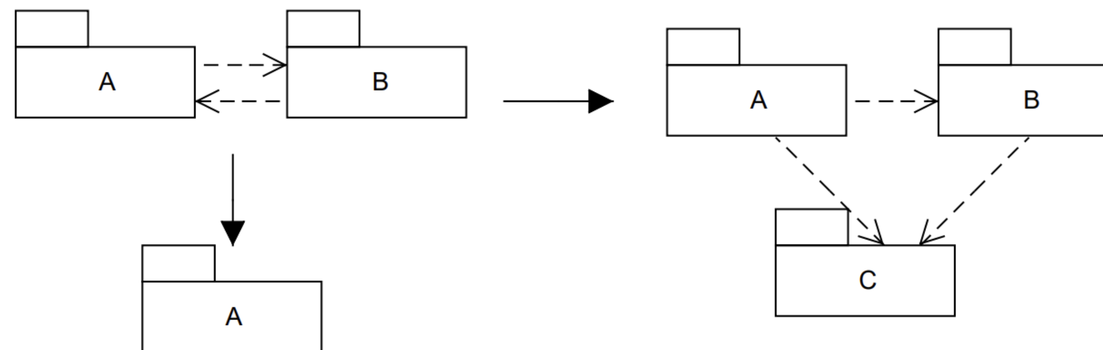
- Merge – „A PackageMerge is a directed relationship between two Packages that indicates that the contents of the target mergedPackage are combined into the source receivingPackage
- The «merge» connector indicates that the Controller Package's elements have been imported into GenApply, including Controller's nested and imported contents.
- If an element already exists within GenApply, such as Loader and Time, these elements' definitions are expanded by those included in the Package Controller. All elements added or updated by the merge are noted by a generalization relationship back to that Package.“



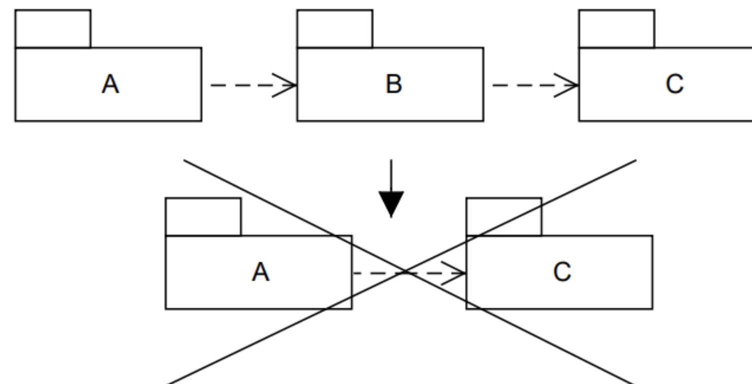
UML

diagram balíkov
poznámky

- Problém cirkulárnej závislosti



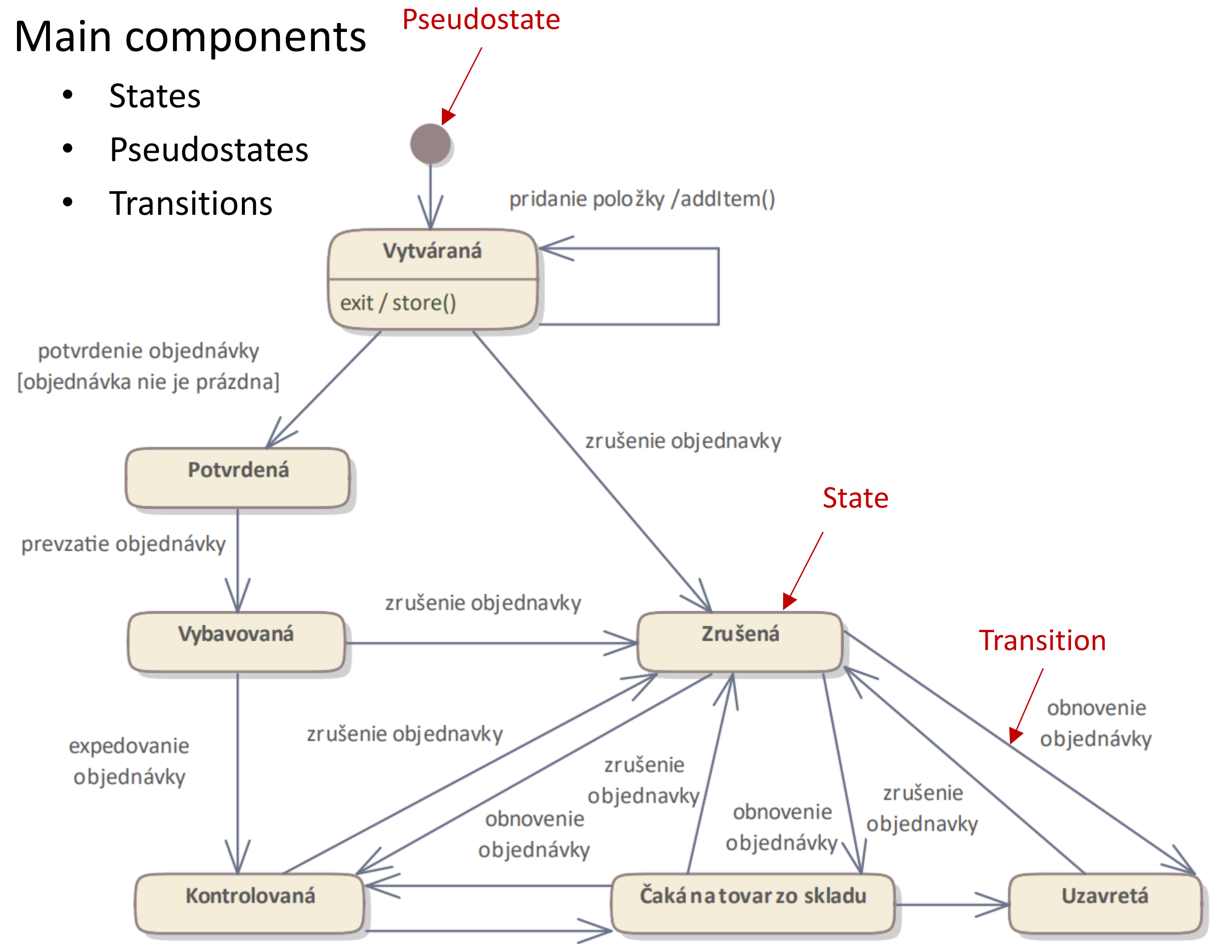
- Vzťah závislosti medzi balíkmi nie je tranzitívny



UML

stavový diagram

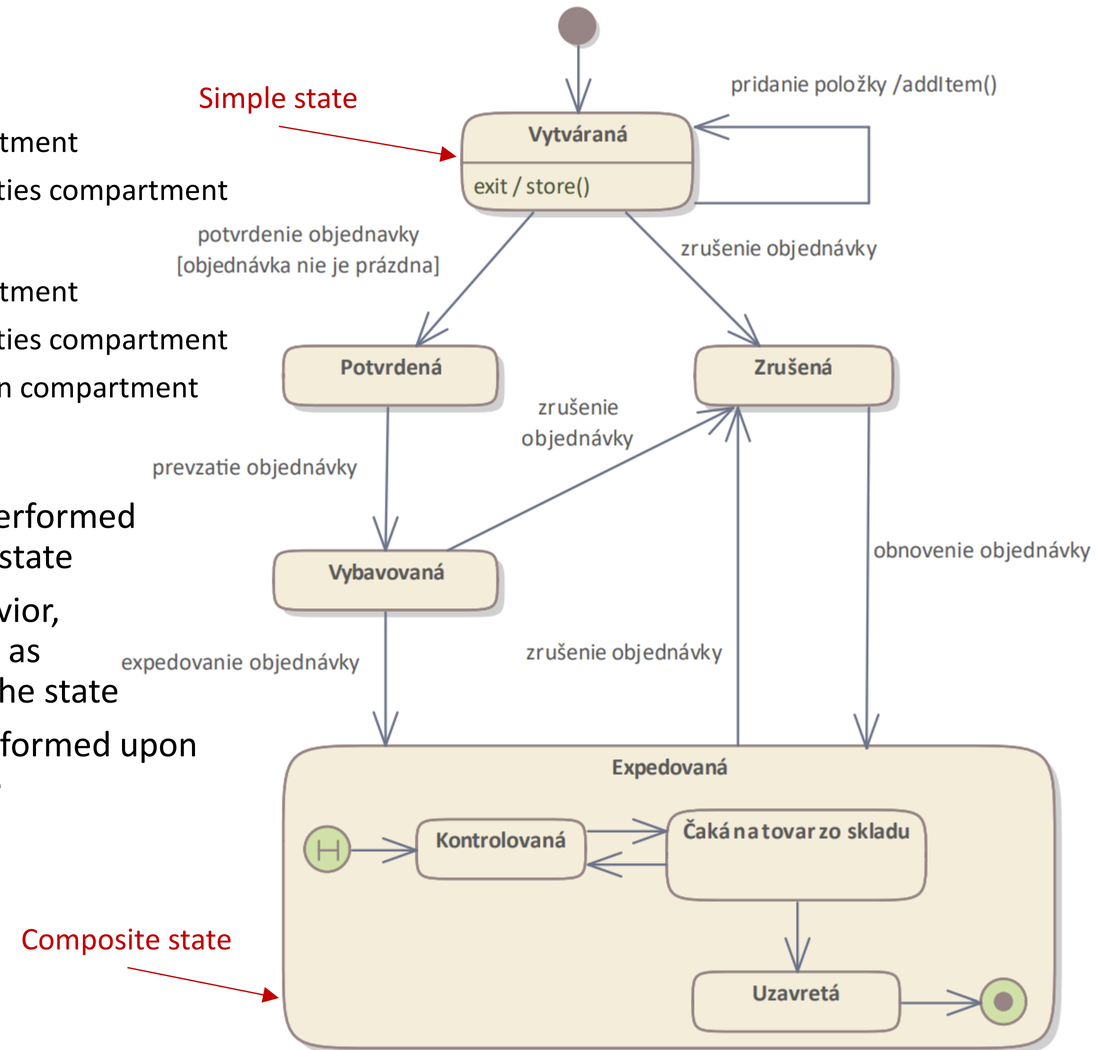
- **State machine diagram -** behavior diagram which shows discrete behavior of a part of designed system through finite state transitions.
- Main components
  - States
  - Pseudostates
  - Transitions



UML

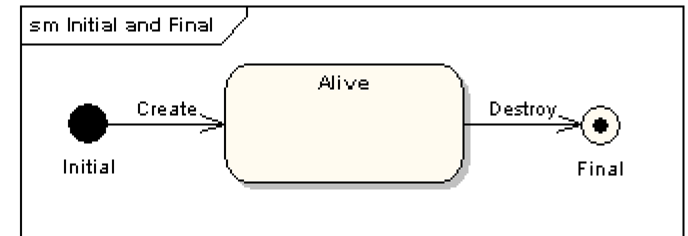
stavový diagram

- **State** in behavioral state machines models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some behavior
- Kinds of states:
  - Simple state
    - name compartment
    - internal activities compartment
  - Composite state
    - name compartment
    - internal activities compartment
    - decomposition compartment
- Internal activities
  - entry - behavior performed upon entry to the state
  - do - ongoing behavior, performed as long as the element is in the state
  - exit - behavior performed upon exit from the state



UML
stavový diagram

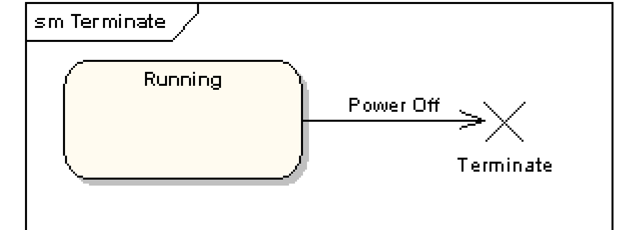
- **Pseudostate** - abstract vertex that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths.
- Kinds of pseudostates:
  - **Initial pseudostate** - represents a default vertex that is the source for a single transition to the default state of a composite state. There can be *at most one initial vertex* in a region. The *outgoing transition* from the initial vertex *may have a behavior, but not a trigger or guard*.
  - Terminate pseudostate
  - Entry point
  - Exit point
  - Choice
  - Join
  - Fork
  - Junction
  - Shallow history pseudostate
  - Deep history pseudostate



UML

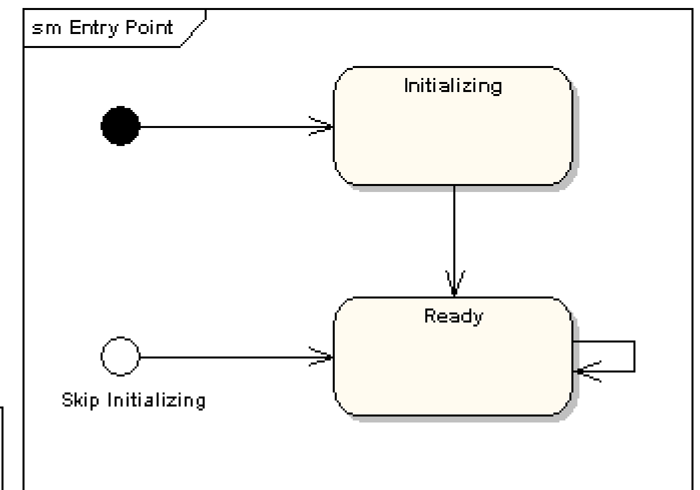
stavový diagram

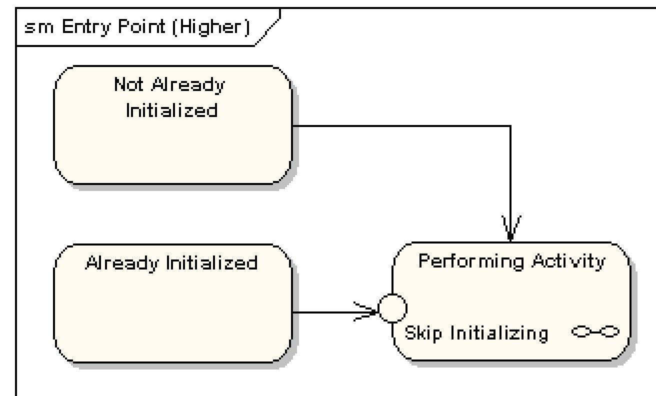
- **Pseudostate** - abstract vertex that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths.
- Kinds of pseudostates:
  - Initial pseudostate
  - **Terminate pseudostate** - implies that the execution of this state machine by means of its context object is terminated. The state machine does not exit any states nor does it perform any exit actions other than those associated with the transition leading to the terminate pseudostate. Entering a terminate pseudostate is equivalent to invoking a DestroyObjectAction.
  - Entry point
  - Exit point
  - Choice
  - Join
  - Fork
  - Junction
  - Shallow history pseudostate
  - Deep history pseudostate



UML
stavový diagram

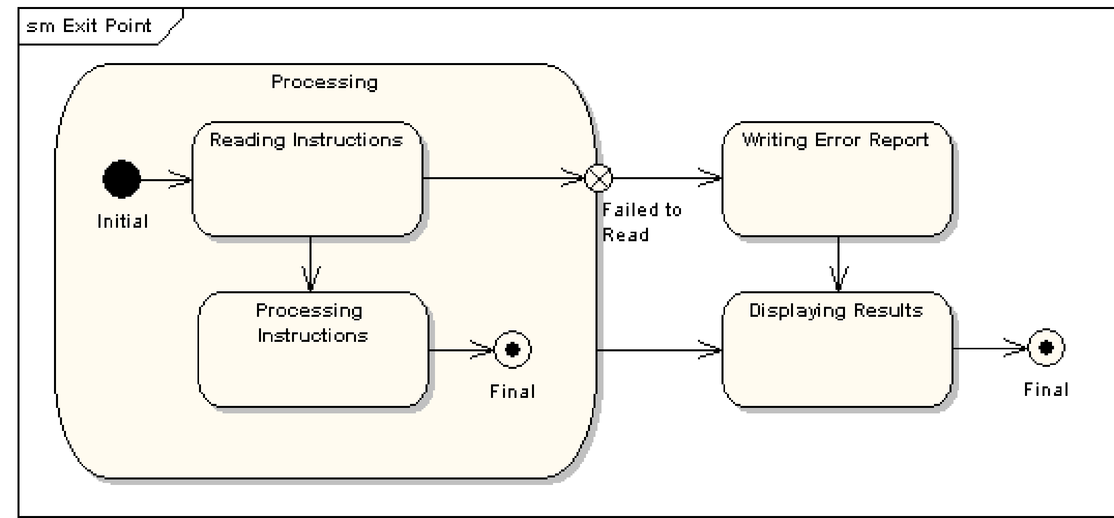
- **Pseudostate** - abstract vertex that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths.
- Kinds of pseudostates:
  - Initial pseudostate
  - Terminate pseudostate
  - **Entry point** - an entry point of a state machine or composite state. Sometimes you don't want to enter a sub-machine at the normal initial state.
  - Exit point
  - Choice
  - Join
  - Fork
  - Junction
  - Shallow history pseudostate
  - Deep history pseudostate





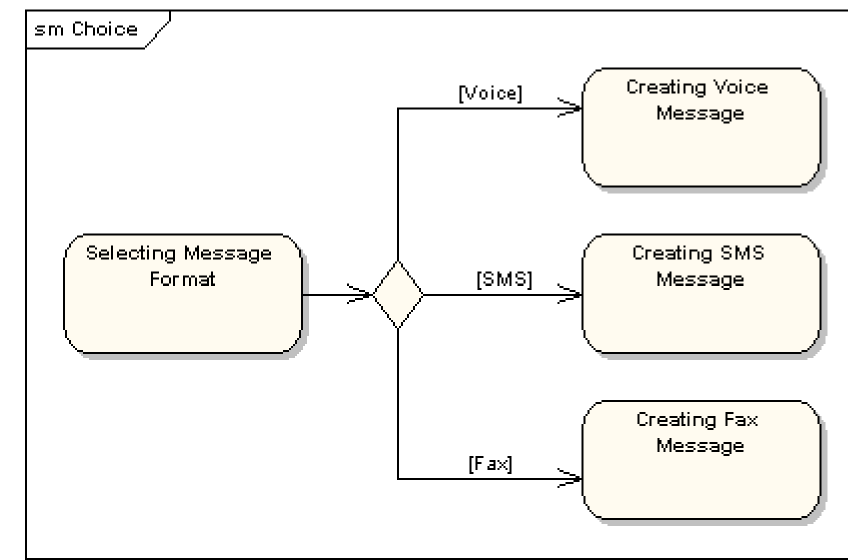
UML
stavový diagram

- **Pseudostate** - abstract vertex that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths.
- Kinds of pseudostates:
  - Initial pseudostate
  - Terminate pseudostate
  - Entry point
  - **Exit point** - exit point of a state machine or composite state
  - Choice
  - Join
  - Fork
  - Junction
  - Shallow history pseudostate
  - Deep history pseudostate



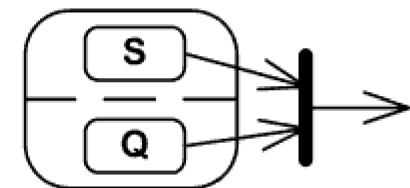
UML
stavový diagram

- **Pseudostate** - abstract vertex that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths.
- Kinds of pseudostates:
  - Initial pseudostate
  - Terminate pseudostate
  - Entry point
  - Exit point
  - **Choice -** realizes a dynamic conditional branch. It evaluates the guards of the triggers of its outgoing transitions to select only one outgoing transition. The following diagram shows that whichever state is arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state (dynamic).
  - Join
  - Fork
  - Junction
  - Shallow history pseudostate
  - Deep history pseudostate



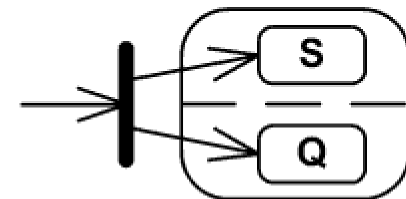
UML
stavový diagram

- **Pseudostate** - abstract vertex that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths.
- Kinds of pseudostates:
  - Initial pseudostate
  - Terminate pseudostate
  - Entry point
  - Exit point
  - Choice
  - **Join -** merges several transitions originating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards or triggers.
  - Fork
  - Junction
  - Shallow history pseudostate
  - Deep history pseudostate



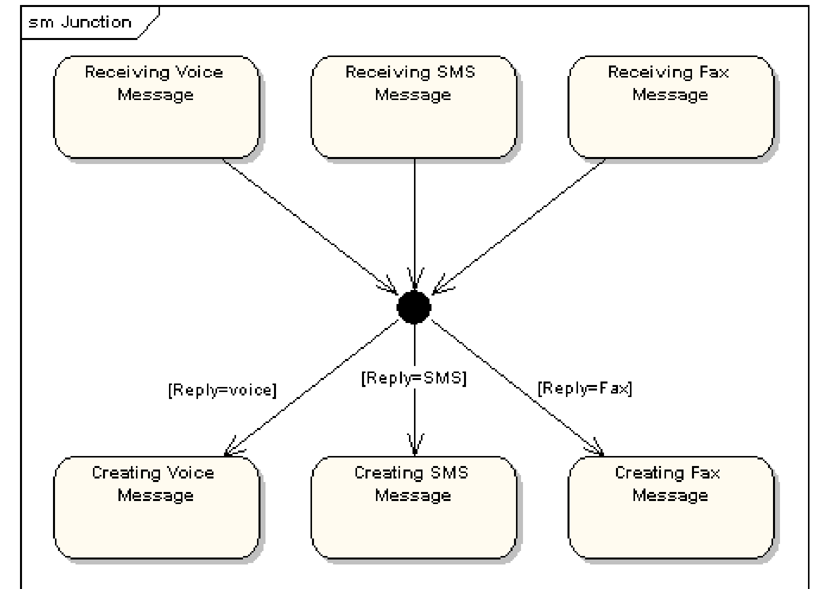
UML
stavový diagram

- **Pseudostate** - abstract vertex that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths.
- Kinds of pseudostates:
  - Initial pseudostate
  - Terminate pseudostate
  - Entry point
  - Exit point
  - Choice
  - Join
  - **Fork -** serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices (i.e., vertices in different regions of a composite state). The segments outgoing from a fork vertex must not have guards or triggers.
  - Junction
  - Shallow history pseudostate
  - Deep history pseudostate



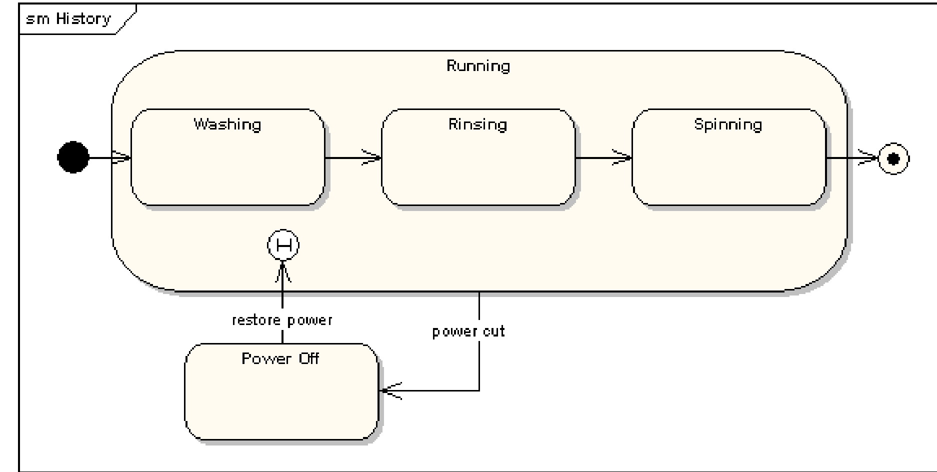
UML
stavový diagram

- **Pseudostate** - abstract vertex that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths.
- Kinds of pseudostates:
    - Initial pseudostate
    - Terminate pseudostate
    - Entry point
    - Exit point
    - Choice
    - Join
    - Fork
    - **Junction -** used to chain together multiple transitions. A single junction can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition. Junctions are semantic-free. A junction which splits an incoming transition into multiple outgoing transitions realizes a static conditional branch, as opposed to a choice pseudo-state which realizes a dynamic conditional branch.
    - Shallow history pseudostate
    - Deep history pseudostate



UML
stavový diagram

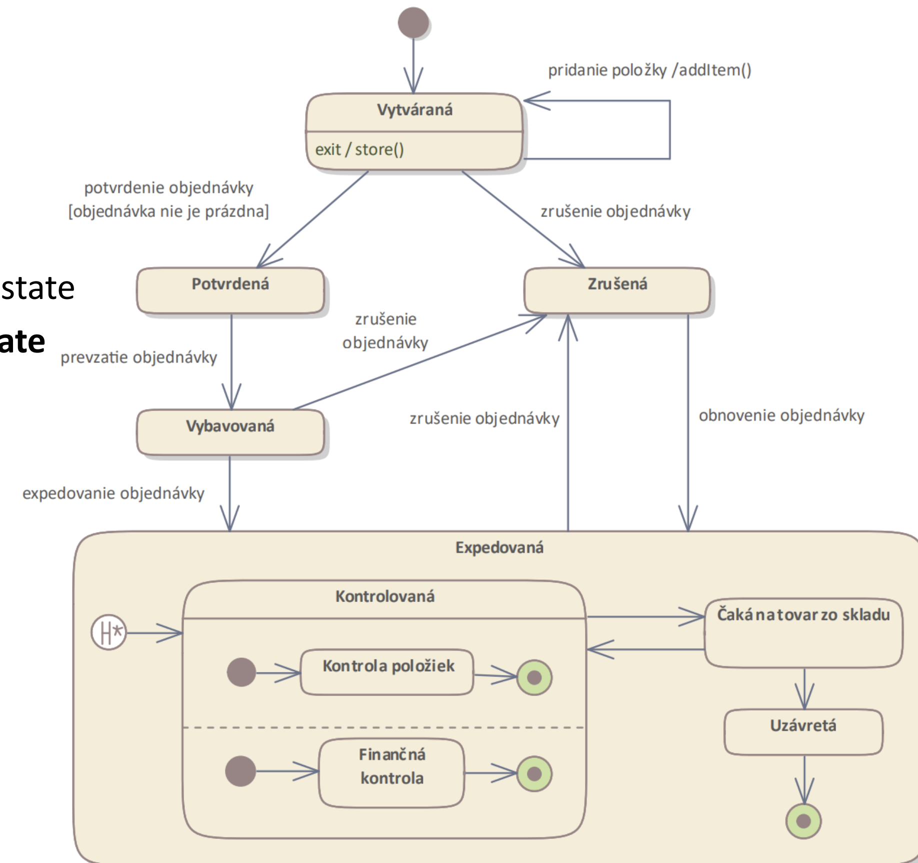
- **Pseudostate** - abstract vertex that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths.
- Kinds of pseudostates:
  - Initial pseudostate
  - Terminate pseudostate
  - Entry point
  - Exit point
  - Choice
  - Join
  - Fork
  - Junction
  - **Shallow history pseudostate** - used to remember the previous state of a state machine when it was interrupted.
  - Deep history pseudostate



UML

stavový diagram

- **Pseudostate** - abstract vertex that encompasses different types of transient vertices in the state machine graph. Pseudostates are typically used to connect multiple transitions into more complex state transitions paths.
- Kinds of pseudostates:
  - Initial pseudostate
  - Terminate pseudostate
  - Entry point
  - Exit point
  - Choice
  - Join
  - Fork
  - Junction
  - Shallow history pseudostate
  - **Deep history pseudostate**



UML

stavový diagram

- **Transition** - directed relationship between a source vertex and a target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type.

transition ::= [*triggers*] [*guard*] ['/' *behavior-expression*]
triggers ::= *trigger* [',' *trigger*]*
guard ::= '[' *constraint* ']'

- **Trigger** - the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time.
- **Guard** - a condition which must be true in order for the trigger to cause the transition.
- **Effect / behavior-expression** - action which will be invoked directly on the object that owns the state machine as a result of the transition.