UML

Unified Modeling Language

- 1995 počátek
- 1997 verze 1.0 leden
- dnes verze 2.0 (vývoj stále nedokončen)
- Standardní notace
- OMG
- podpora velkých firem (Microsoft, IBM, Oracle, HP…)
- popisuje ***struktury***
- popisuje ***chování***

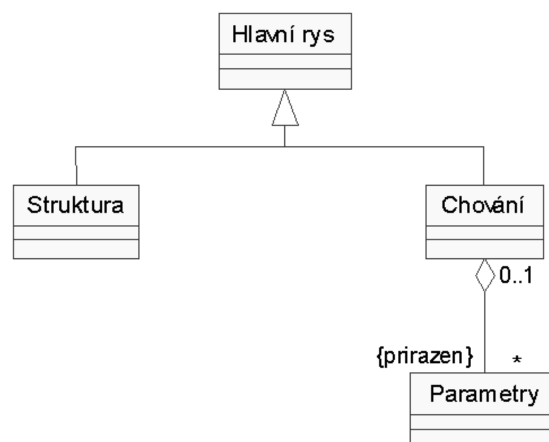
Součásti UML

Celá metodologie je definována několika dobře napsanými dokumenty.

- UML Summary - dokument obsahující stručný úvod do cílů a zdrojů metodologie.
- UML Semantics - dokument definující sémantiku UML ze tří pohledů
- Abstraktní syntaxe
- Dobře navržených pravidel
- Sémantiky
- UML Notation Guide - dokument popisující grafickou notaci UML
- UML Extensions - dokumenty popisující extense základního modelu, současné době existují dvě následující
- UML Extension for Objectory Process for Software Engineering
- UML Extension for Business Modeling
- Object Constraint Language Specifikation - dokument popisující formální jazyk použitý v UML

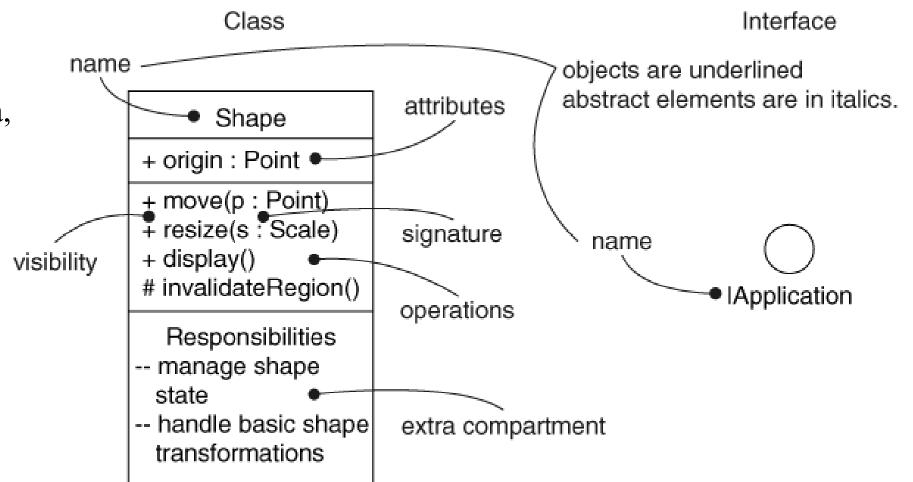
Metamodel UML

- Metamodel je diagram, obvykle class diagram, který definuje notaci (tedy UML)
- Příklad části metamodelu jazyka UML ukazující vztah mezi asociacemi a generalizací



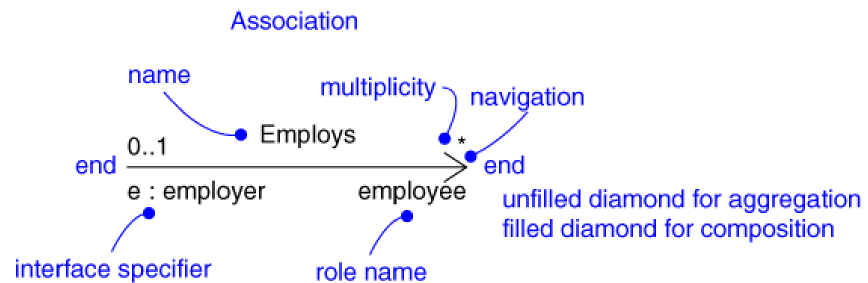
Prvky modelů

- Prvky struktury
- třídy, rozhraní, spolupráce, use case, aktivní třída, komponenta, uzel
- Prvky chování
- interakce, stavový diagram
- Prvky skládání
- package, subsystém
- Další prvky



Vztahy

- Závislost
- Asociace
- Dědičnost
- Realizace



Pravidla UML

UML poskytuje pravidla pro

- pojmenování
- rozsah platnosti
- rozsah viditelnosti
- integritní omezení
- provedení modelu
- Specifikace
- za grafickou notací jsou ukryty další informace
- sémantika modelu je založena na definovaném metamodelu

Diagramy

- Diagram je pohled na model
  - prezentován z pozice určitého uživatele
  - poskytuje určitou reprezentaci systému
  - sémanticky konzistentní s dalšími pohledy
- V rámci UML je definováno devět standardních diagramů
- statický pohled: ***use case***, ***třídy***, ***objekty***, ***komponenty***, ***rozmístění***
- dynamické pohledy: ***sekvence***, ***spolupráce***, ***stavy***, ***aktivity***

Use case (model jednání / případ užití / kontextový diagram...)

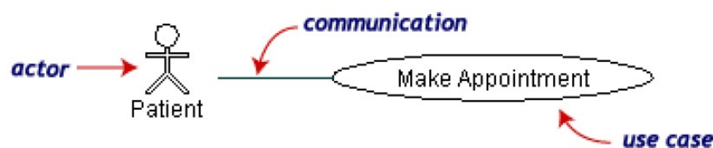
Use case diagrams describe what a system does from the standpoint of an external observer. The emphasis is on *what* a system does rather than *how.*

Use case diagrams are closely connected to scenarios. A **scenario** is an example of what happens when someone interacts with the system. Here is a scenario for a medical clinic.

> "A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot. "

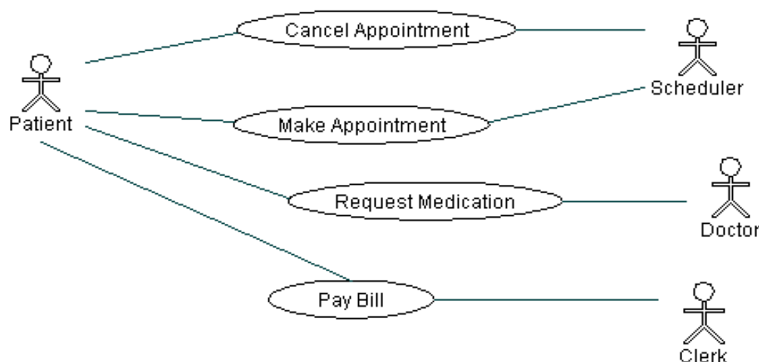
A **use case** is a summary of scenarios for a single task or goal. An **actor** is who or what initiates the events involved in that task. Actors are simply roles that people or objects play. The picture below is a **Make**

Appointment use case for the medical clinic. The actor is a **Patient**. The connection between actor and use case is a **communication association** (or **communication** for short).



Actors are stick figures. Use cases are ovals. Communications are lines that link actors to use cases.

A use case diagram is a collection of actors, use cases, and their communications. We've put **Make Appointment** as part of a diagram with four actors and four use cases. Notice that a single use case can have multiple actors.



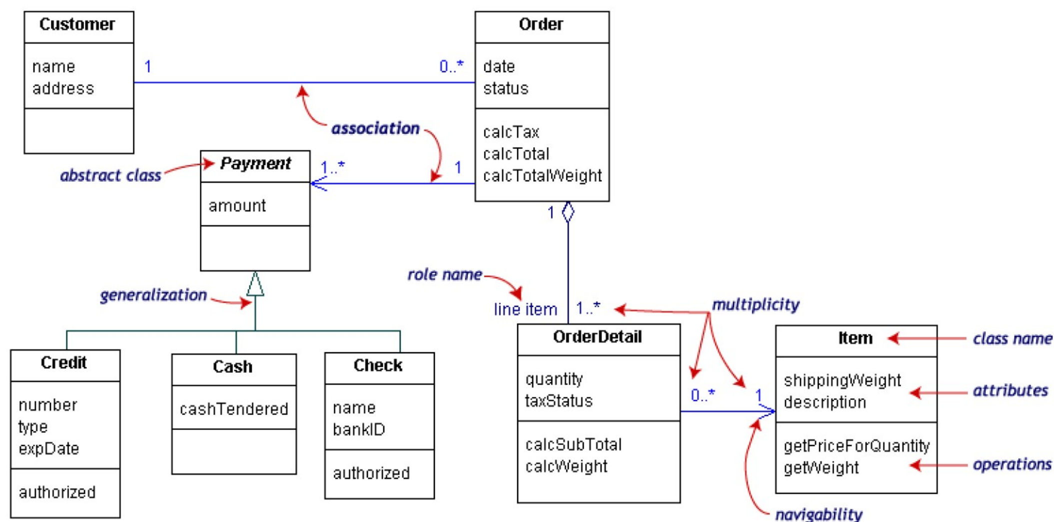
Use case diagrams are helpful in three areas.

- **determining features (requirements)**. New use cases often generate new requirements as the system is analyzed and the design takes shape.
- **communicating with clients**. Their notational simplicity makes use case diagrams a good way for developers to communicate with clients.
- **generating test cases**. The collection of scenarios for a use case may suggest a suite of test cases for those scenarios.

Class diagram (diagramy tříd)

A Class diagram gives an overview of a system by showing its classes and the relationships among them. Class diagrams are static -- they display what interacts but not what happens when they do interact.

The class diagram below models a customer order from a retail catalog. The central class is the **Order**. Associated with it are the **Customer** making the purchase and the **Payment**. A **Payment** is one of three kinds: **Cash**, **Check**, or **Credit**. The order contains **OrderDetails** (line items), each with its associated **Item**.



UML class notation is a rectangle divided into three parts: class name, attributes, and operations. Names of abstract classes, such as ***Payment***, are in italics. Relationships between classes are the connecting links.

Our class diagram has three kinds of relationships.

- **association** -- a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes.
- **aggregation** -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole. In our diagram, **Order** has a collection of **OrderDetails**.
- **generalization** -- an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass. ***Payment*** is a superclass of **Cash**, **Check**, and **Credit**.

An association has two ends. An end may have a **role name** to clarify the nature of the association. For example, an **OrderDetail** is a line item of each **Order**.

A **navigability** arrow on an association shows which direction the association can be traversed or queried. An **OrderDetail** can be queried about its **Item**, but not the other way around. The arrow also lets you know who "owns" the association's implementation; in this case, **OrderDetail** has an **Item**. Associations with no navigability arrows are bi-directional.

The **multiplicity** of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers. In our example, there can be only one **Customer** for each **Order**, but a **Customer** can have any number of **Orders**.

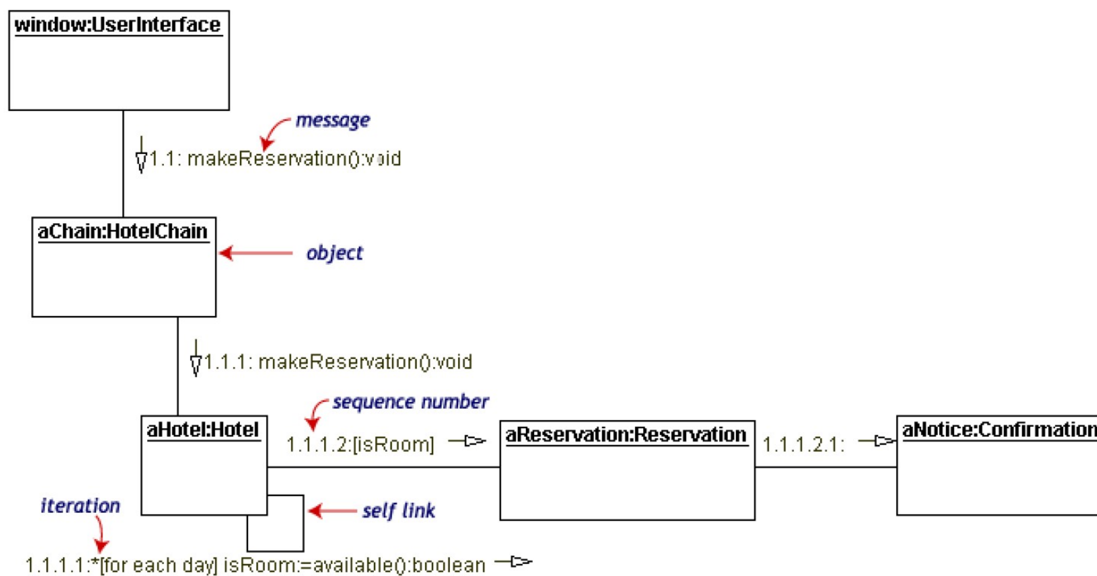
This table gives the most common multiplicities.

Multiplicities	Meaning
0..1	zero or one instance. The notation ***n . . m*** indicates ***n*** to ***m*** instances.
0..* *or* *****	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

Every class diagram has classes, associations, and multiplicities. Navigability and roles are optional items placed in a diagram to provide clarity.

Collaboration diagrams (diagramy spolupráce)

Collaboration diagrams are also interaction diagrams. They convey the same information as sequence diagrams, but they focus on object roles instead of the times that messages are sent. In a sequence diagram, object roles are the vertices and messages are the connecting links.



The object-role rectangles are labeled with either class or object names (or both). Class names are preceded by colons (:).

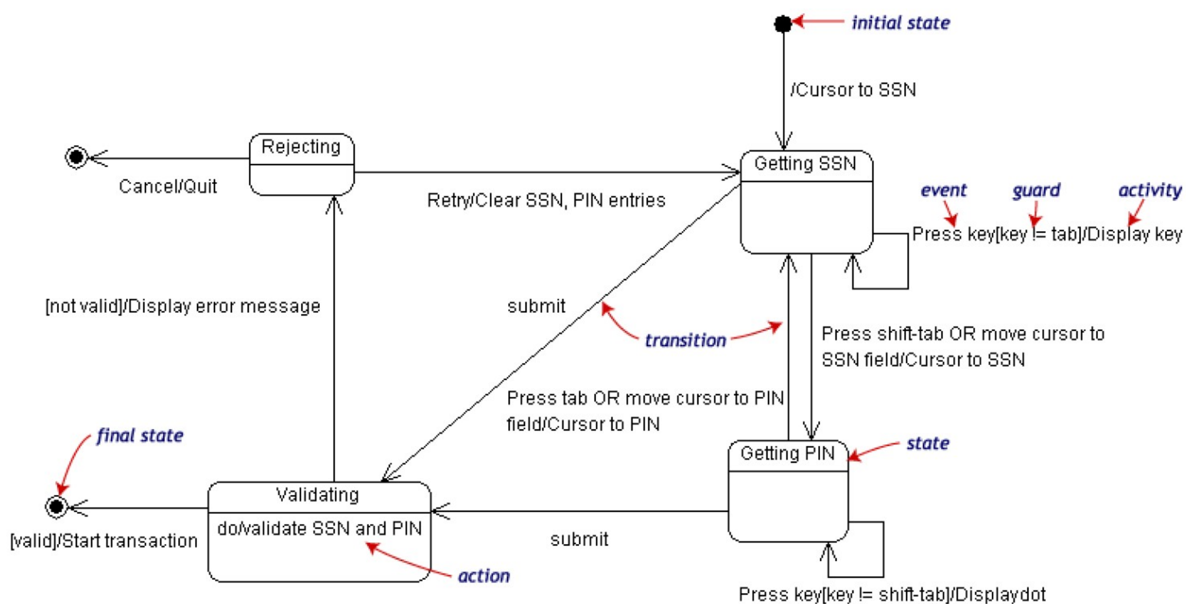
Each message in a collaboration diagram has a **sequence number**. The top-level message is numbered 1. Messages at the same level (sent during the same call) have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.

Statechart diagrams (stavové diagramy)

Objects have behaviors and state. The state of an object depends on its current activity or condition. A **statechart diagram** shows the possible states of the object and the transitions that cause a change in state.

Our example diagram models the login part of an online banking system. Logging in consists of entering a valid social security number and personal id number, then submitting the information for validation.

Logging in can be factored into four non-overlapping states: **Getting SSN**, **Getting PIN**, **Validating**, and **Rejecting**. From each state comes a complete set of **transitions** that determine the subsequent state.



States are rounded rectangles. Transitions are arrows from one state to another. Events or conditions that trigger transitions are written beside the arrows. Our diagram has two self-transition, one on **Getting SSN** and another on **Getting PIN**.

The initial state (black circle) is a dummy to start the action. Final states are also dummy states that terminate the action.

The action that occurs as a result of an event or condition is expressed in the form /action While in its **Validating** state, the object does not wait for an outside event to trigger a transition. Instead, it performs an activity. The result of that activity determines its subsequent state.

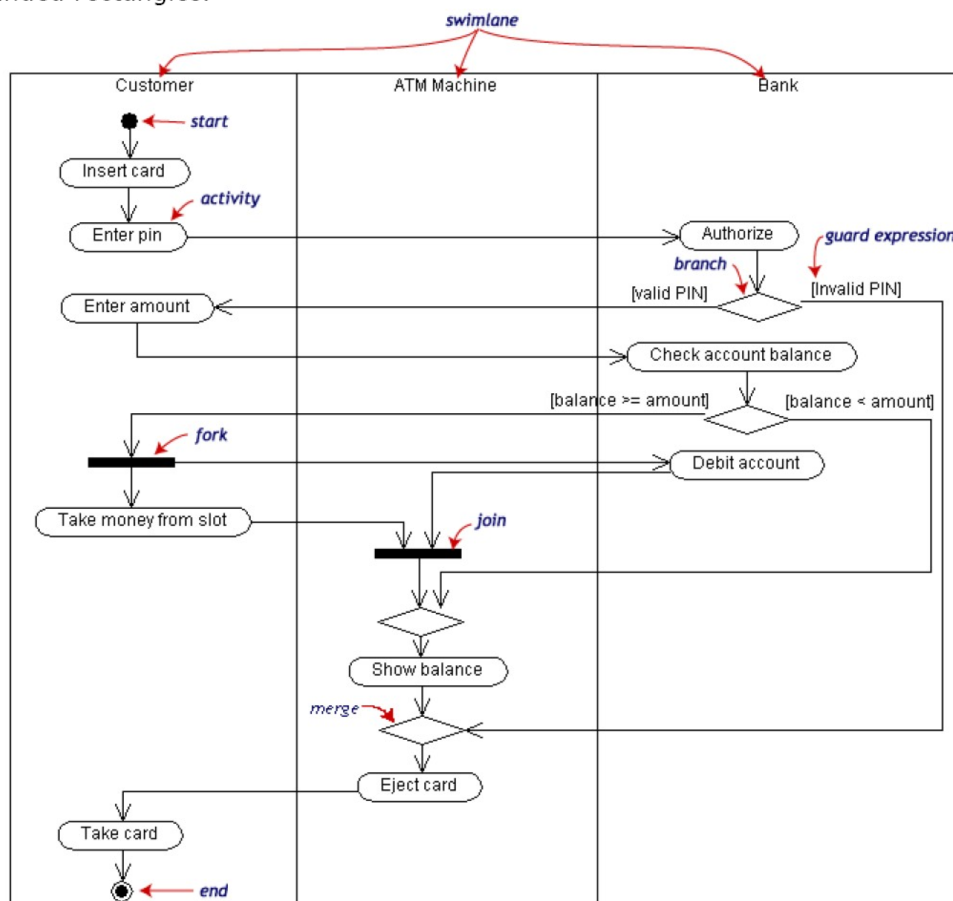
Activity diagram (diagramy aktivit)

An **activity diagram** is essentially a fancy flowchart. Activity diagrams and statechart diagrams are related. While a statechart diagram focuses attention on an object undergoing a process (or on a process as an object), an activity diagram focuses on the flow of activities involved in a single process. The activity diagram shows the how those activities depend on one another.

For our example, we used the following process.

"Withdraw money from a bank account through an ATM."

The three involved classes (people, etc.) of the activity are **Customer**, **ATM**, and **Bank**. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are rounded rectangles.



Activity diagrams can be divided into object **swimlanes** that determine which object is responsible for which activity. A single **transition** comes out of each activity, connecting it to the next activity.

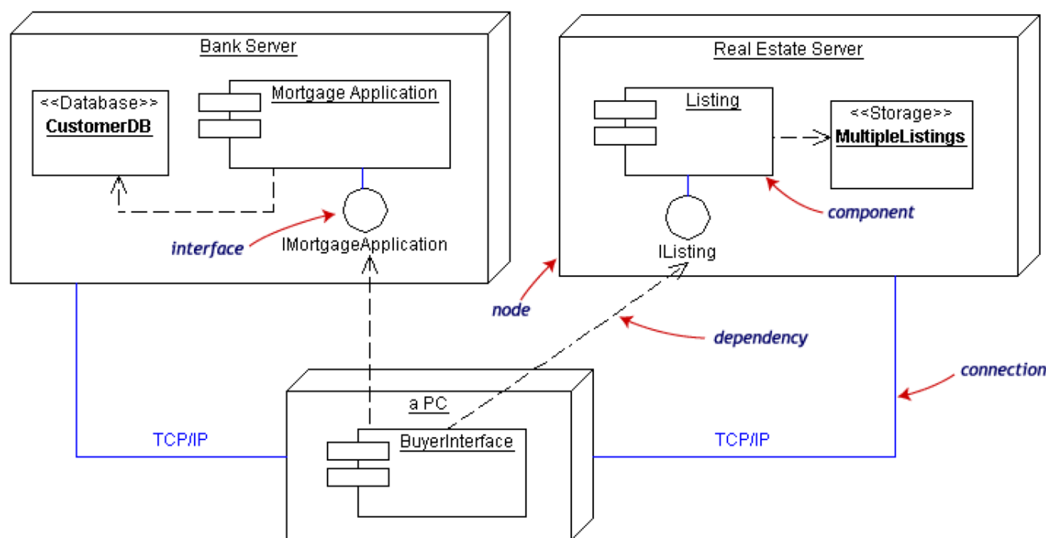
A transition may **branch** into two or more mutually exclusive transitions. **Guard expressions** (inside [ ]) label the transitions coming out of a branch. A branch and its subsequent **merge** marking the end of the branch appear in the diagram as hollow diamonds.

A transition may **fork** into two or more parallel activities. The fork and the subsequent **join** of the threads coming out of the fork appear in the diagram as solid bars.

Component and Deployment diagrams (diagramy komponent a diagramy nasazení)

A **component** is a code module. Component diagrams are physical analogs of class diagram. **Deployment diagrams** show the physical configurations of software and hardware.

The following deployment diagram shows the relationships among software and hardware components involved in real estate transactions.



The physical hardware is made up of **nodes**. Each component belongs on a node. Components are shown as rectangles with two tabs at the upper left.

OCL - Object Constraint Language Specification

Většina modelovacích technik nedokáže všechny závislosti mezi jednotlivými konstrukty vyjádřit pouhým grafickým znázorněním a musí si pomáhat popisem. Slovní popis však není vždy jednoznačný, proto byl pro použití v UML vyvinut speciální formální jazyk ***OCL*** (*Object Constraint Language)*, jednoduchý pro zápis i čtení. Zdrojem toho jazyka byl obchodní modelovací jazyk interně používaný ve firmě IBM.

OCL není programovací jazyk, je to jazyk určený pro modelovací techniky. Jedná se však o typový jazyk. K jakému účelu je ho možné v UML použít? Například pro specifikování podmínek vykonání operací či metod, pro specifikování invariantů tříd, jako navigační jazyk, pro definování constraint, pokud například poznámka v UML nestačí.

OCL umí pracovat s množinami objektů, například s collection, set, bag, sequence. Jako ukázku vyjadřovacích schopností jazyka OCL si uvedeme výraz, který má vyjádřit podmínku, že všechny instance osoby mají rozdílné jméno.

```
Person.allInstances?forAll(p1, p2 | p1 <> p2 implies p1.name <> p2.name)
```

Person.allInstances je set (množina) všech osob a je typu Set(Person). Nad touto množinou je aplikována operace forAll, která vykoná definovanou činnost pro všechny prvky množiny. Parametrem činnosti jsou dva prvky množiny, pro které platí, že pokud jsou rozdílné, mají rozdílná jména. Je vidět, že definování podmínky je deklarativní a nikoliv algoritmické.

Dalším příkladem je podmínka, která vybere všechny zaměstnance jejichž věk, je větší než v 50 let.

```
self.emloyee.select(p : Person | p.age > 50)
```

Tento výraz čteme následovně. Na množinu zaměstnanců aktivní třídy je uplatněna operace výběru (select), která vybíra prvky splňující podmínku, že věk zaměstnance je větší než hodnota 50.

Dalším příkladem je výraz jehož výsledkem je množina všech rozdílných dat narození všech osob.

```
self.emloyee?collect(birthDate)?asSet
```

Na množinu zaměstnanců je uplatněna operace collect, která vytvoří množinu všech dat narození, nad touto množinou je dále uplatněna operace asSet, která z této množiny vytvoří množinu všech rozdílných dat narození všech zaměstanců.

Doufám, že výše uvedené ukázky naznačí možnosti jazyka OCL, více než pokus o neucelený popis jeho syntaxe a sémantiky. Osobně s tím jazykem namám vůbec žádné zkušenosti, takže za příklady neručím.