Java
Lecture 8 - UML

PRAGMATIC IT Learning & Outsourcing Center

Lector: Peter Manolov
Skype: nsghost1
E-mail: p.manolov@gmail.com

www.pragmatic.bg

2013 – 2014

Summary

- UML Basics
- Example of RUP (Rational Unified Process)
- UML Syntax
  - Use Cases
  - Class Diagrams
  - Sequence Diagram

UML Basics

- What is UML
  - UML stands for Unified Modeling Language
  - It was created to centralize the different modeling approaches that existed.
  - UML is the successor of the torrent of all of the design and analysis methods that were created in the late 80's and early 90's .
  - UML is just a language not a method
    - A method consists of a language and a process.

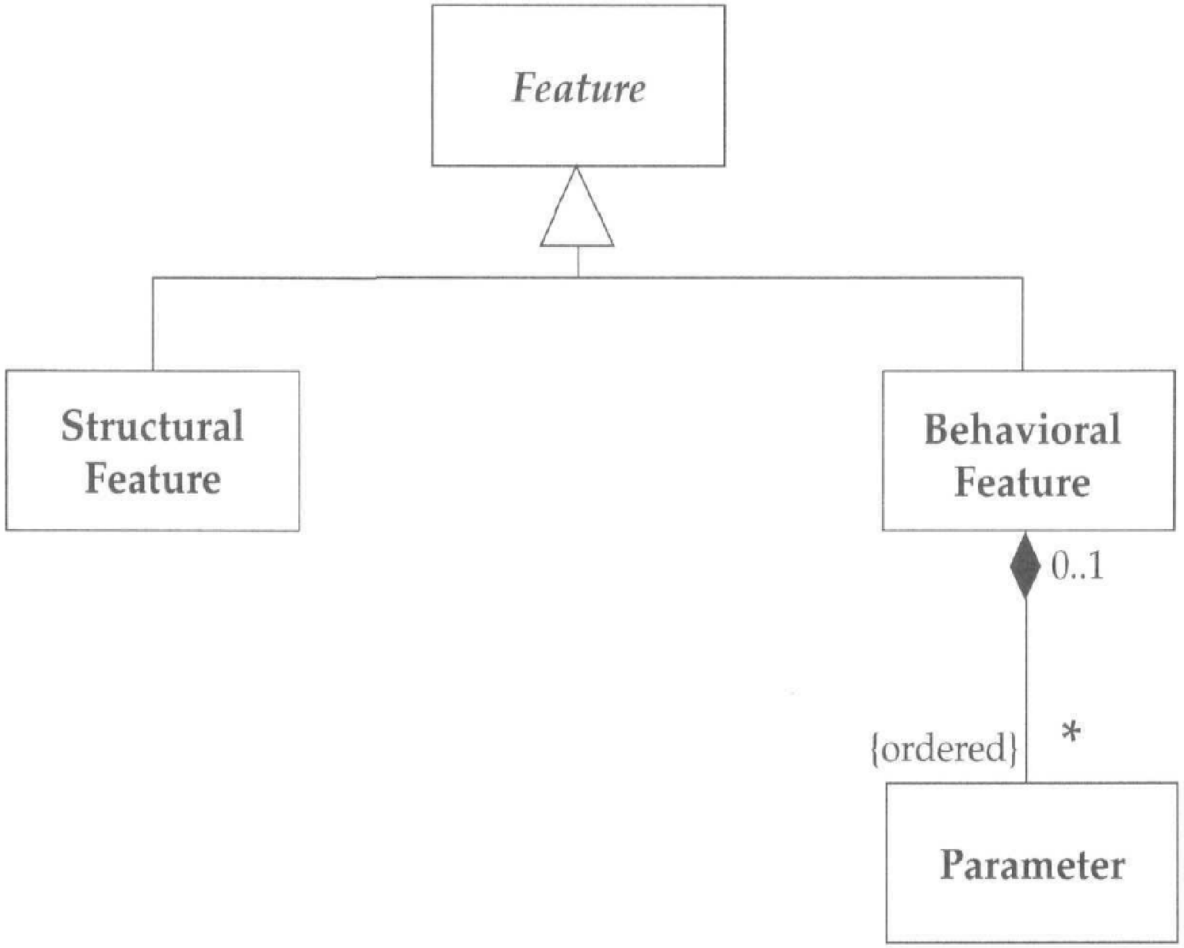
UML Basics

- UML is the graphical notations used to express design ideas and decisions.
- It's a key communication tool
- Helps determine key areas of the system while still in the development phase of a project

Meta data

- **Meta data** is data about the data
- By describing the contents and context of data files, the quality of the original data/files is greatly increased.
- Example : Think of a library and it it's books. The data is in the books, however if you wish the understand how to use the data, how to find a book, you need the registry and card catalogs. The latter is the library metadata.

UML Meta Model

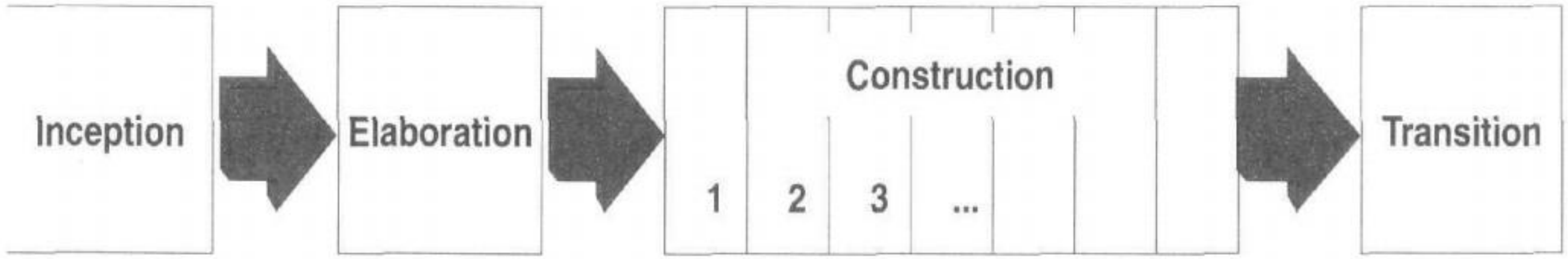


UML Metamodel

- Why one should abide by the metamodel
  - Although it is a good idea to fully understand how the UML language is structured it is not necessary to fully understand it in order for you to use it.
  - The primary goal of UML is to act as a conversation tool, to send the information across, one need not be too rigorous with the UML meta model to be able to do "your job".
  - One must abide by the general principals of the UML meta model and by expressing his/her's ideas in a logical and intuitive way , in fact UML was created with the idea to be intuitive, it's not necessary to follow all the "strict" rules that there are. Remember UML is meant to communicate ideas, therefore it must be used in a way that those ideas are send across as easily as possible.

Rational Unified Process

- An example of an iterative process
  - It means that the software is not released as a huge piece of code rather as a sum of releases.
- Iterative process are created so that risks are managed better.



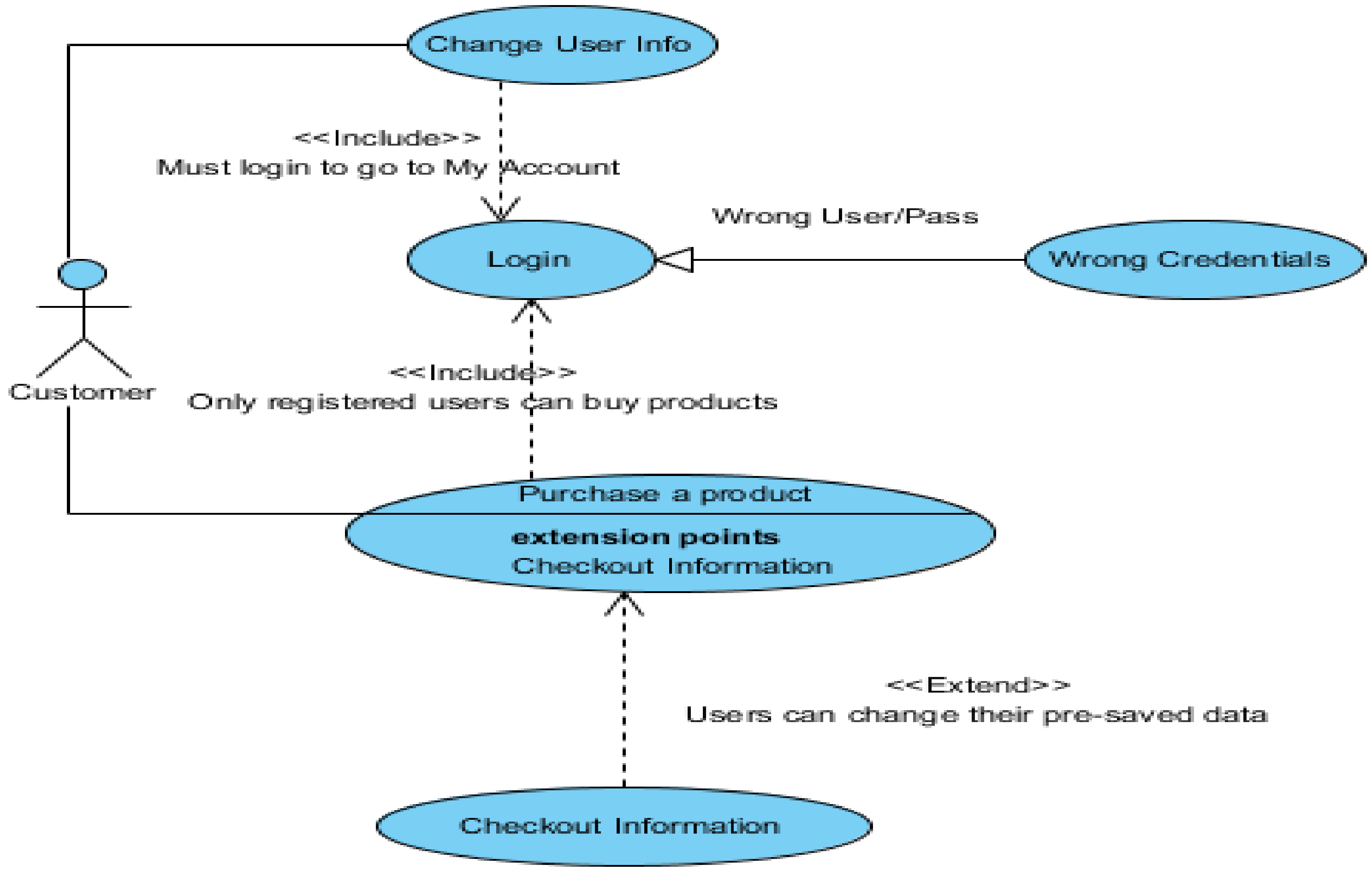
Rational Unified Process

- Inception
  - Here the business (for whom we're building the system) decide what exactly they need and whether to invest in the software or not
- Elaboration
  - Here more detailed information is gather. I high level design is created. Baseline architecture starts to take place. Plans for the construction phase begin to take form.

Rational Unified Process

- Construction
  - Here the engineers take front stage and start building the low level design. With each new iteration the projects comes closer to the business requirements gathered in the inception and elaboration phases.
- Transition
  - Here some of the extra work can take place. Beta test, Integration tests, Stress Tests, Performance test. Special user training if necessary. This is the phase where the clients “get” their application.

Use Cases


Change User Info
<<Include>>
Must login to go to My Account
Login
Wrong User/Pass
Wrong Credentials
Customer
<<Include>>
Only registered users can buy products
Purchase a product
extension points
Checkout Information
<<Extend>>
Users can change their pre-saved data
Checkout Information

Use Cases

- Actor
  - A role the “user” plays in respect to the system. In the diagram above that would be the “customer”. An actor can be either a “real human” or a thirdparty system or process. An actor is an element of the process that usually triggers it or benefits from it.
- Usecase
  - Describing the actual process

Use case relationships

- Include relationship
  - As the name suggests this is used when you have a behavior which can be part of a larger process. Sometimes such a sub-process can take part in different larger processes. In the example a customer can buy a product , but first must be logged in therefore the login process must take place as part of the purchase (checkout) procedure. The customer however can change their personal information (the My Account pages), however access to that information takes places only after a successful login. The *login* use case is the same and part of two different larger processes , *product purchase* and *access to my account*

Use case relationships

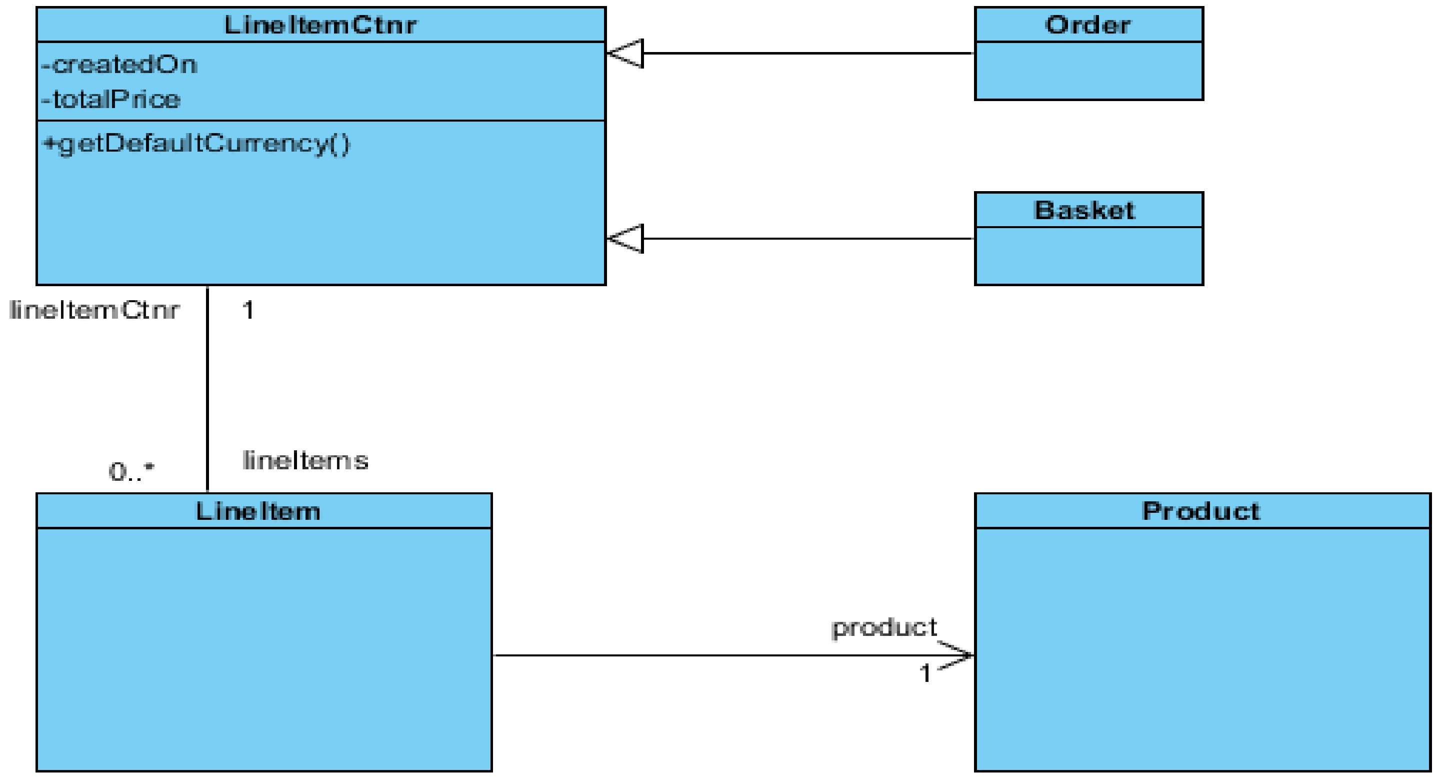
- Generalization
  - Generalization is used when you have similar behavior across different use cases. In effect its used to describe alternative scenarios. In our example a login can be either successful or unsuccessful, both need to be address however they are similar, because both take place on the login page and require the user to be presented with the login form, however based on the user input we have two different user scenarios.

- Extend
  - Extend is very similar to generalization, however it provides better control over how the additional , overriding behavior is attached to the base case. A base case can be extended only if **extension points** are provided. An extension point defines where exactly the new behavior plugs in. In our example a customer may perform the purchase of product however said customer can also change their shipping address , for example, change their shipping address, during the checkout process.

Use case

- When to use use cases relationships ?
  - **Include** when you are repeating yourself in two or more separate use cases thus avoiding repetion
  - **Generalization** when you are describing a variation on normal behavior, an alternative logical path.
  - **Extend** when you are describing a variation on normal behavior and you wish to use the more controlled form, declaring your extension points in your base use case. Helping better identify where the base case scenario will deviate from the best case scenario

Class Diagrams



Class Diagrams

- Class diagrams can be viewed in three ways
  - Conceptual – The ideas , the concepts within the domain currently being studied. In our example, we have the concept of an Order , of a Customer. Concepts although eventually will relate to a class, there is no direct mapping. This is perhaps the most high level perspective a class diagram can take. It doesn't concern itself with the method in which the ideas presented will be implemented thus giving a certain freedom over the actual technology (language) that eventually will be used.

Class Diagrams

- Specification
  - This is the actual software , however it's more abstract view. The interfaces, the contracts we initially specify.
- Implementation
  - Here we have the actual classes and implementations. The skeleton of our domain classes can be shown here.

Class Diagram - Associations

- Associations
  - This is how we represent the manner in which different classes would relate to one another. Each association is represented by the end of the line which connects two classes. The end can be named, you can put a label there, also, knows as the *role.*
  - An association end also has multiplicity. Multiplicity also acts as a constraint on how much of a certain type there can be within a relationship
    - 1 – a single
    - 0..1 – one or zero
    - 0..* - zero or many
    - 1..* - one or many
    - n..m – from n to m

Class Diagram - Associations

- Associations
  - Associations can also demonstrate how a relationship is traversed. That is represented with an arrow. The class that has the arrow pointing at it, is the child within the relationship meaning that the other class, being the owner, has a handle with a certain constraint on it. (Here handle means a class property). For example – A LineItem has a Product, it's a 1..1 relationship, however the product doesn't know about the lineItem it is in, as a product can be part of many LineItems but only one a time.
  - If there is no arrow it is generally assumed that the relationship is bi-directional, meaning both classes can see each other. (Orders have lineItems, lineItems know about their order)

Class Diagrams - Attributes

- Attributes are very similar to associations. They can still represent an association between different classes.
- Depending on the "flavor" of UML used an attribute can specify

```
visibility name:type=defaultValue
```

Class Diagram - Operations

- An operation is the action a class would carry out. It’s the class’ methods.

```
visibility name(parameter-list): return-type-expression {property-string}
```

- Visibility is **+**(public), **#**(protected), or **-** (private)
- Name is a *string*
- parameter-list containscomma-separated parameters whose syntax is similar to that for attributes: name:type = defaultvalue.
- return-type-expression is a comma-separated list of return types

  Example :

```
+getTotal(currency:Currency):Money.
```

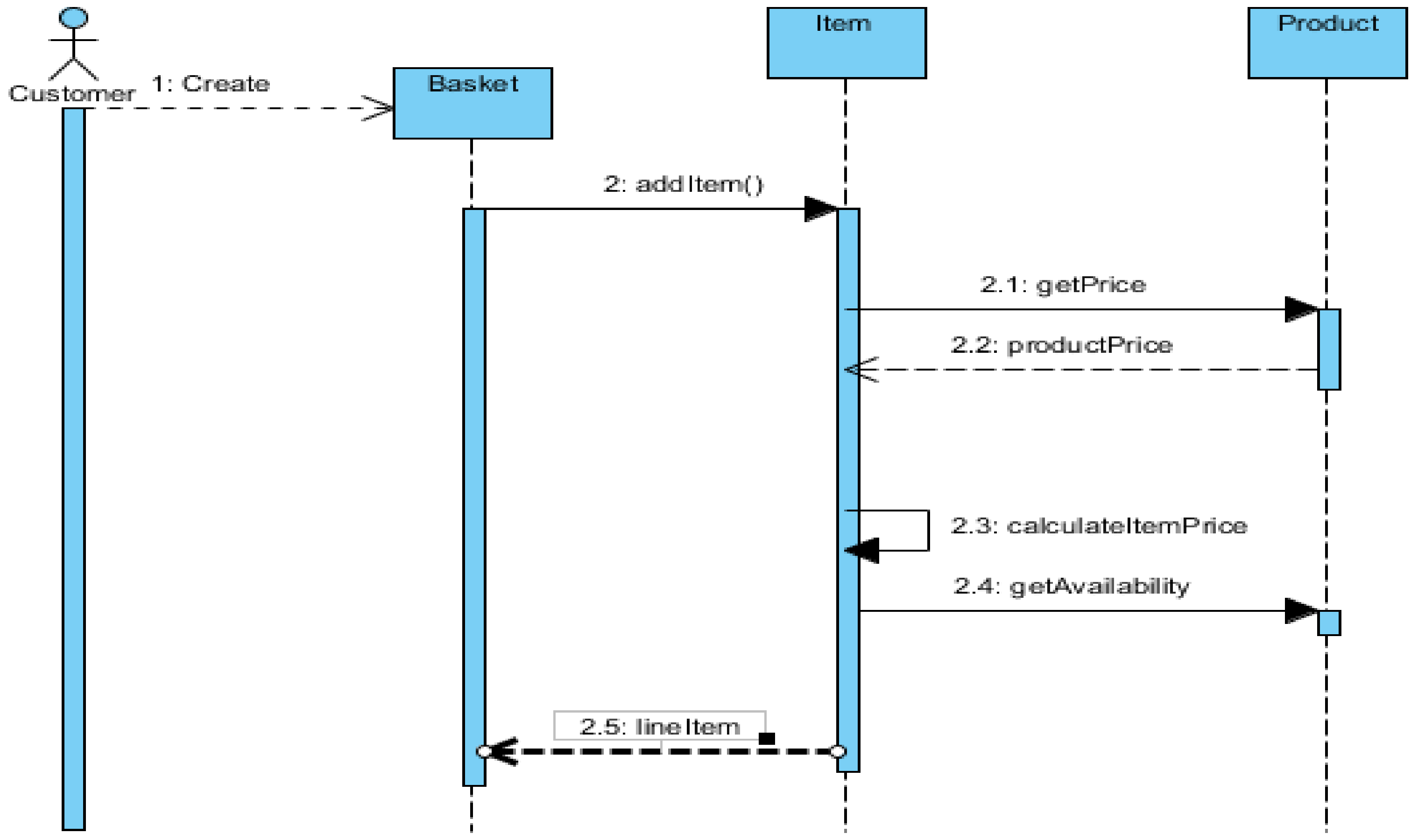
Class Diagrams - Generalization

- Generalization – An example of generalization is the connection between a LineItemCtnr and a Basket. For all intent and purposes that is inheritance. It is represented with the an arrow the shaft of which is at the child and the point is at the parent. The arrow point is a hollow triangle.

When to use Class Diagrams

- Class diagrams are perhaps the most central part of any OO design , therefore it is safe to assume that this is the type of diagram most commonly used.
- Don’t overdo them. Class diagrams have many aspects, remember UML is meant to communicate ideas , find the best way to send your across and use only those aspects of class diagrams that you find necessary. Don’t use everything.

Sequence Diagrams



Sequence Diagrams

- Sequence diagrams describe how a group of objects interact with each other
- Usually one sequence diagram per use case.
- In sequence diagrams objects are represented with a box and line coming out of it. That line is called the lifeline of that object. The line represents how long that object "lives" within the context of that interaction.
- The arrow between the lifelines is called a message.

Sequence diagram

- There are self-messages. A message that an objects sends to itself. (example *calculateItemPrice*)
- A condition can be expressed. A message can be send only if a certain criteria is met. That is achieved by putting the condition within brackets **[<cond>]**
- Sometimes we need to have a certain message be executed multiple times, for example iterating over a collection. That is done by using the multiplicity sign *

Sequence Diagrams

- Sequence diagrams are a great tool to visual the interaction between the different classes of our application .
- Sequence diagrams demonstrate the flow of control between the different objects.

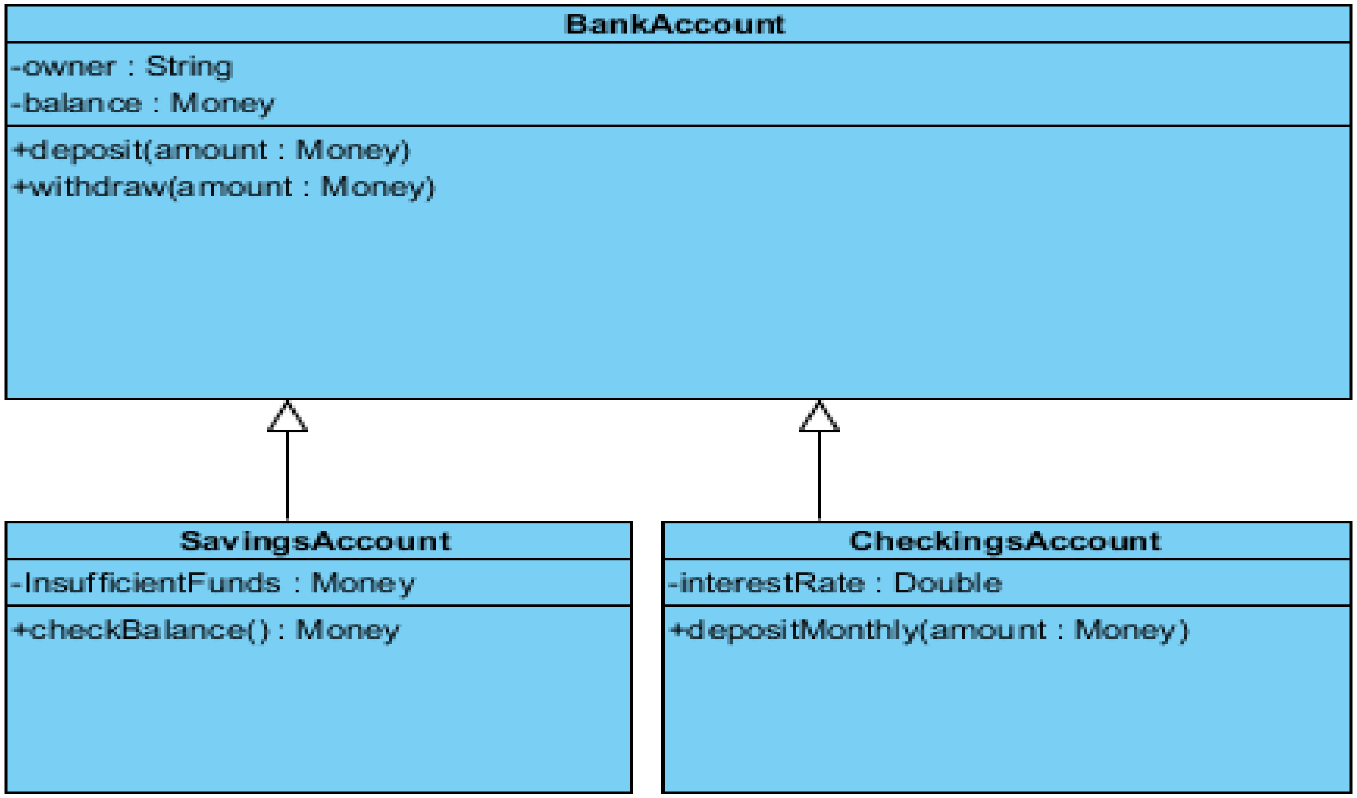
UML and programming

- UML doesn't translate 1 to 1 to any programming languages.
- UML is too abstract to bind it to one development paradigm
- UML's goal is to describe ideas and process and how different components might interact.
- It's up to the developers reading the UML diagrams to decide how they should go about implementing the actual functionality

UML and programming

- There are tools (such as Visual Paradigm) that can generate actual code based on the UML diagrams
- The generated code however usually is just a boiler plate and rarely conforms with your companie's coding standards.

fig. 1


BankAccount
-owner : String
-balance : Money
+deposit(amount : Money)
+withdraw(amount : Money)
SavingsAccount
-InsufficientFunds : Money
+checkBalance() : Money
CheckingsAccount
-interestRate : Double
+depositMonthly(amount : Money)

Problems ?!

- What is the reason behind the creation of UML ?
- What is UML's primary purpose ?
- What is the relationship between UML and programming languages.
- Can I write my program in UML ?
- Look at the fig.1 . Please write the corresponding java code. What kind of diagram is this ?