CS 545: Software Design Patterns
Week 2

Varun Dutt

School of Computing and Electrical Engineering
School of Humanities and Social Sciences
Indian Institute of Technology Mandi, India

Scaling the Heights

UNIFIED MODELING LANGUAGE

- The Unified Modelling Language (UML) is a diagramming language or notation to specify, visualize and document models of Object Oriented software systems

- UML is designed specifically for Object Oriented software design

- UML is not a development method, it does not tell you what to do first and what to do next or how to design your system, but helps you visualize your design and communicate with others

UML

In an effort to promote Object Oriented designs, three leading object oriented programming researchers joined ranks to combine their languages:

- Grady Booch (BOOCH)
- Jim Rumbaugh (OML: object modeling technique)
- Ivar Jacobsen (OOSE: object oriented software eng)

and come up with an industry standard [mid 1990's].

UML DIAGRAMS

- The UML elements which represent different parts of a software are used to create diagrams, which represent a certain part, or a point of view of the system

- Ideas can be represented in UML using different types of diagrams such as:
  - **Use Case Diagram (Requirements)**
  - **Class Diagram (Static Design)**
  - **Sequence Diagram (Dynamic Design)**
  - Collaboration Diagram (similar to sequence diagram)
  - State Diagram (states of an object)
  - Activity Diagram (states of an object with simultaneous activities)
  - Component Diagram (how components are placed)
  - Deployment Diagram (how the components are deployed)
  - Entity Relationship Diagram (for DB design)

USE CASE DIAGRAM (Gathering Requirements)

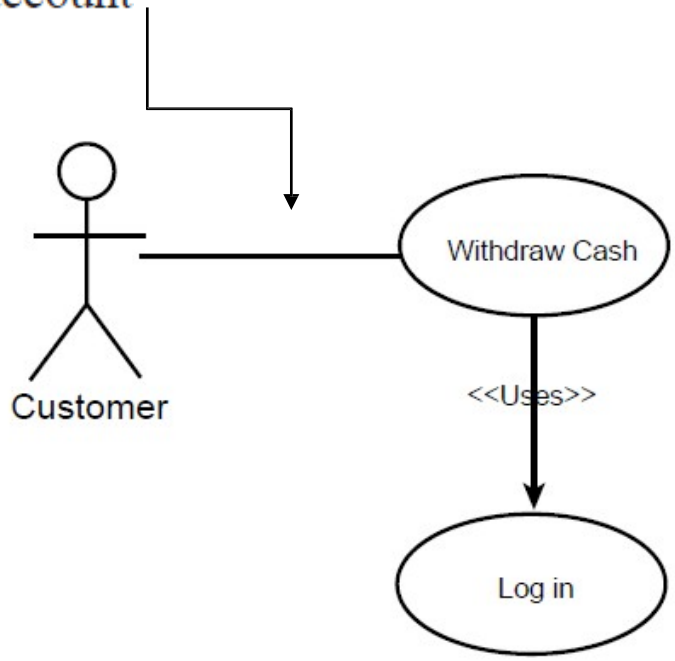
- GOAL
  - A use case is nothing more than a high-level description of what the product is supposed to do.
  - Defines “WHAT” system should do and DOES NOT specify “HOW” it to be achieved
- KEY TERMS
  - Use Case
    - Use Cases are descriptions of the typical interactions between the users of a system and the system itself, used to represent the external interface of the system
  - Actor
    - An actor is an external entity (outside of the system) that interacts with the system by participating (and often initiating) a Use Case. Actors can be in real life people (for example users of the system), other computer systems or external events

USE CASE DIAGRAM (ATM Example)

Examples of Use cases:

- Customer checks his or her balances
- Customer deposits money to his or her account
- Customer withdraws money from his or her account
- Customer transfers money between accounts
- Customer opens an account
- Customer closes an account

The «uses» stereo-type.



Use «uses» when you would otherwise copy and paste the entire use case in place, and use «extends» when you only use the use case under certain definable conditions.

Scenario(s) (for a Use Case)

Use Case:	Customer withdraws cash
Scenario:	Successful cash withdrawal from checking
Preconditions:	Customer is already logged in to system
Trigger:	Customer requests "withdrawal"
Description:	Customer chooses to withdraw cash from a checking account. There is sufficient cash in the account, there is sufficient cash and receipt paper in the ATM, and the network is up and running. The ATM asks the customer to indicate the amount of the withdrawal, and the customer asks for $300, a legal amount to withdraw at this time. The machine dispenses $300 and prints a receipt, and the customer takes the money and the receipt.
PostConditions:	Customer account is debited $300, and customer has $300 cash.

Going from Scenarios to Classes

In any case, the biggest stumbling block for many novices is finding the initial set of classes and understanding what makes a well-designed class. One simplistic technique suggests writing out the use-case scenarios and then creating a class for every noun. Consider the following use-case scenario:

> **Customer** chooses to withdraw **cash** from **checking**. There is sufficient cash in the **account**, sufficient cash and **receipts** in the **ATM**, and the **network** is up and running. The ATM asks the customer to indicate an **amount** for the **withdrawal**, and the customer asks for $300, a legal amount to withdraw at this time. The **machine** dispenses $300 and prints a receipt, and the customer takes the **money** and the receipt.

Customer
Cash
Checking
Account
Receipts

ATM
Network
Amount
Withdrawal
Machine
Money

After pruning →

Customer
Cash (money, amount, withdrawal)
Checking
Account
Receipts
ATM (machine)
Network

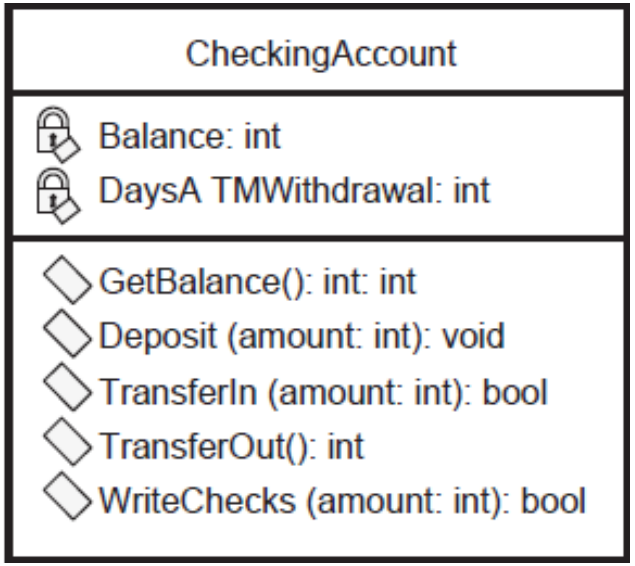
CLASS DIAGRAM (Designing Systems; Static)

- GOAL
  - How different classes that make up system and how they relate to each other
  - Considered “static” because they show the classes, along with their methods and attributes as well as the static relationships between them: which classes “know” each other or which classes “are part” of another class, but do NOT show the method calls between them

- KEY TERMS
  - Class
    - A Class defines the attributes and the methods for a set of objects
    - All objects of this class (instances of this class) share the same behavior, and have the same set of attributes (each object has its own set)

A Class (Example)

A checking account class diagram.

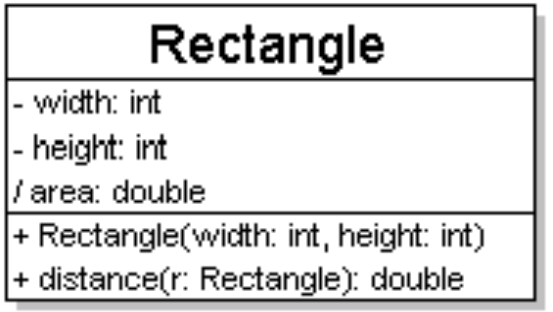


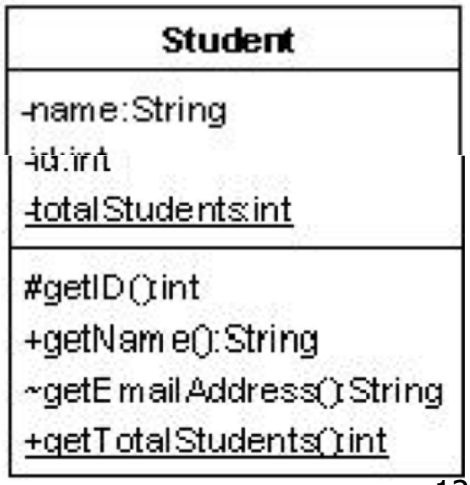
Corresponding Definition of a Class in C++

```
class CheckingAccount
{
public:
    // Construction / Destruction
    CheckingAccount ();
    ~CheckingAccount ();

    // Public operations
    int GetBalance ();
    void Deposit (int amount);
    BOOL TransferIn (int amount);
    int TransferOut ();
    BOOL WriteChecks (int amount);
private:
    // Private attributes
    int Balance;
    int DaysATMWithdrawal;
};
```

Diagram of one class

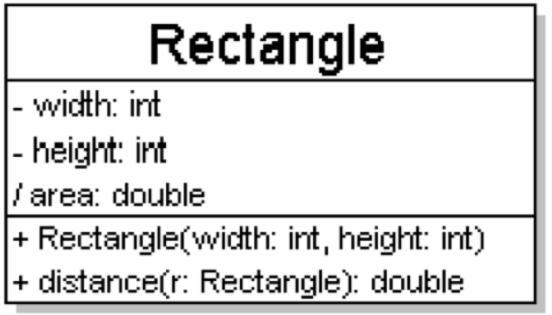
- class name in top of box
  - write <<interface>> on top of interfaces' names
  - use *italics* for an *abstract class* name
- attributes (optional)
  - should include all fields of the object
- operations / methods (optional)
  - may omit trivial (get/set) methods
    - but don't omit any methods from an interface!
  - should not include inherited methods

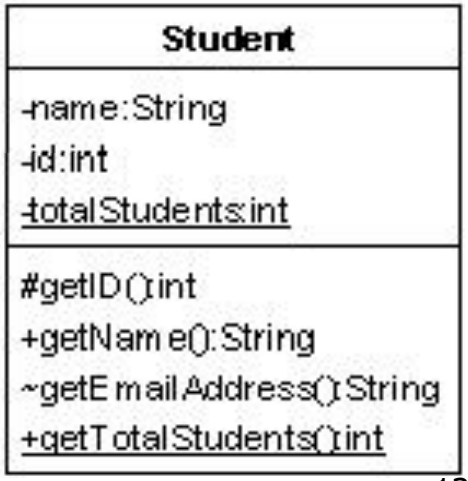




Class attributes

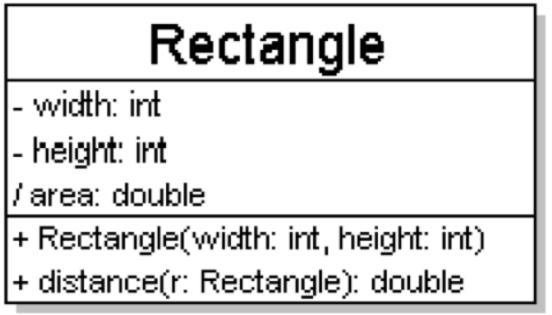
- attributes (fields, instance variables)
  - *visibility name* : *type* [*count*] = *default_value*
  - visibility: + public
    # protected
    - private
    ~ package (default)
    / derived
  - underline static attributes
  - **derived attribute**: not stored, but can be computed from other attribute values
  - attribute example:
    - balance : double = 0.00

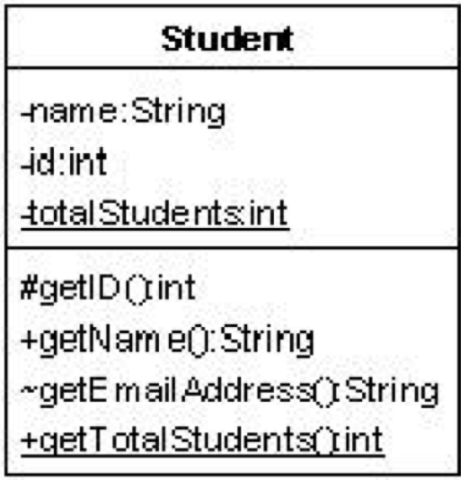




Class operations / methods

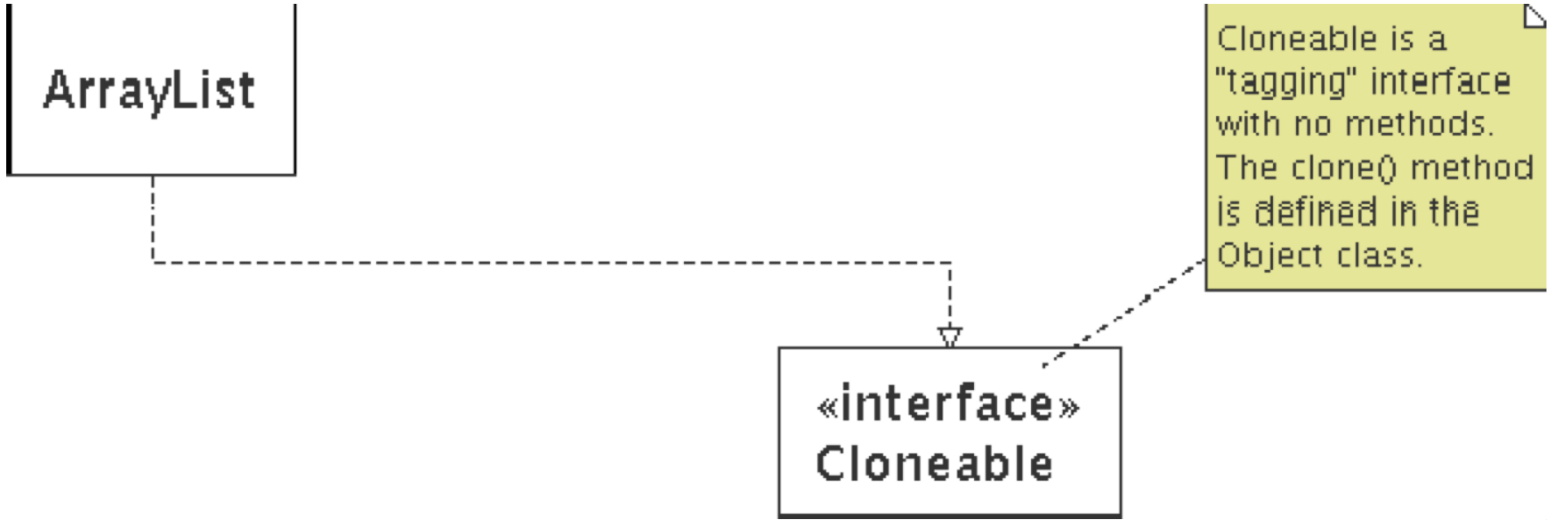
- operations / methods
  - *visibility name* (*parameters*) : *return_type*
  - visibility: + public
    # protected
    - private
    ~ package (default)
  - underline static methods
  - parameter types listed as (name: type)
  - omit *return_type* on constructors and when return type is void
  - method example:
    + distance(p1: Point, p2: Point): double





Comments

- represented as a folded note, attached to the appropriate class/method/etc by a dashed line



Class Diagram (Associations: 1:1)

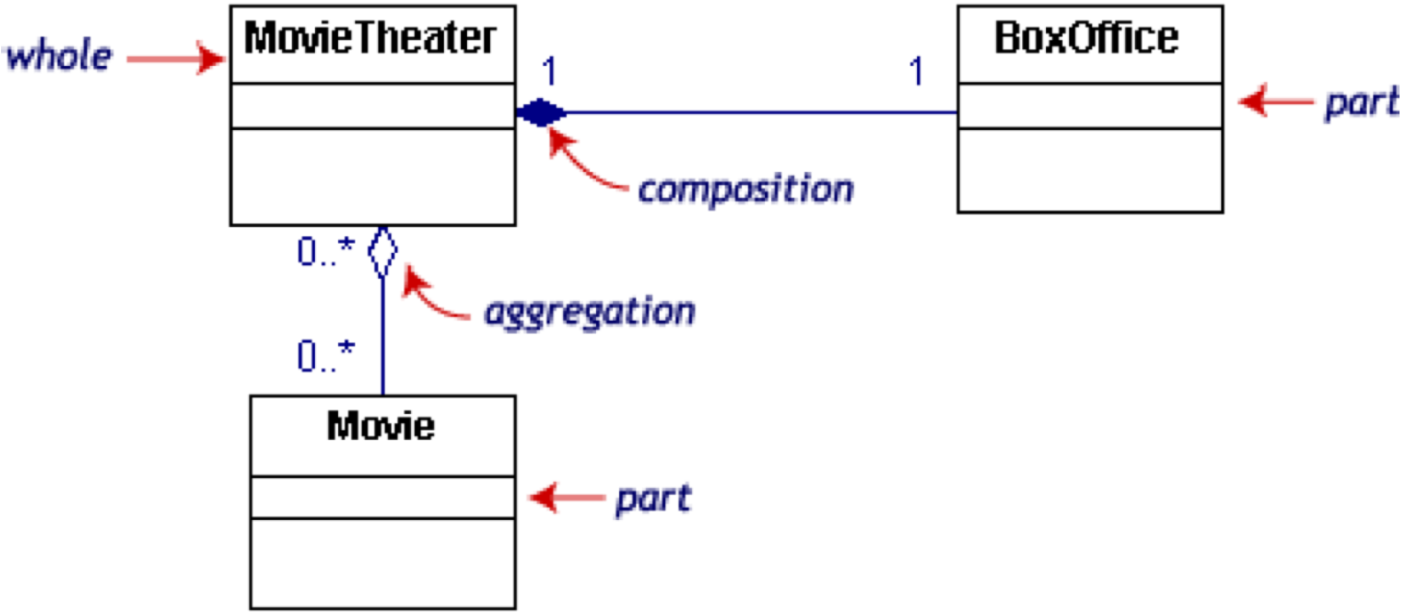
In Code

```
class Renter
{
public:

  Renter ( );
  ~Renter ( );

  Apt * m_dwell;

  void setDwell ( Apt * new_var );

  Apt * getDwell ( );

  Apt GetApt ( )
  {
  }

  void SetApt (Apt A1 )
  {
  }
};
```

```
class Apt
{
public:

  Apt ( );

 ~Apt ( );

  Renter * m_tenant;

  void setTenant ( Renter * new_var );

  Renter * getTenant ( );

  Renter GetRenter ( )
  {
  }

  void SetRenter (Renter R1 )
  {
  }

};
```

Class Diagram (Associations: 1:N)

Composition/aggregation example

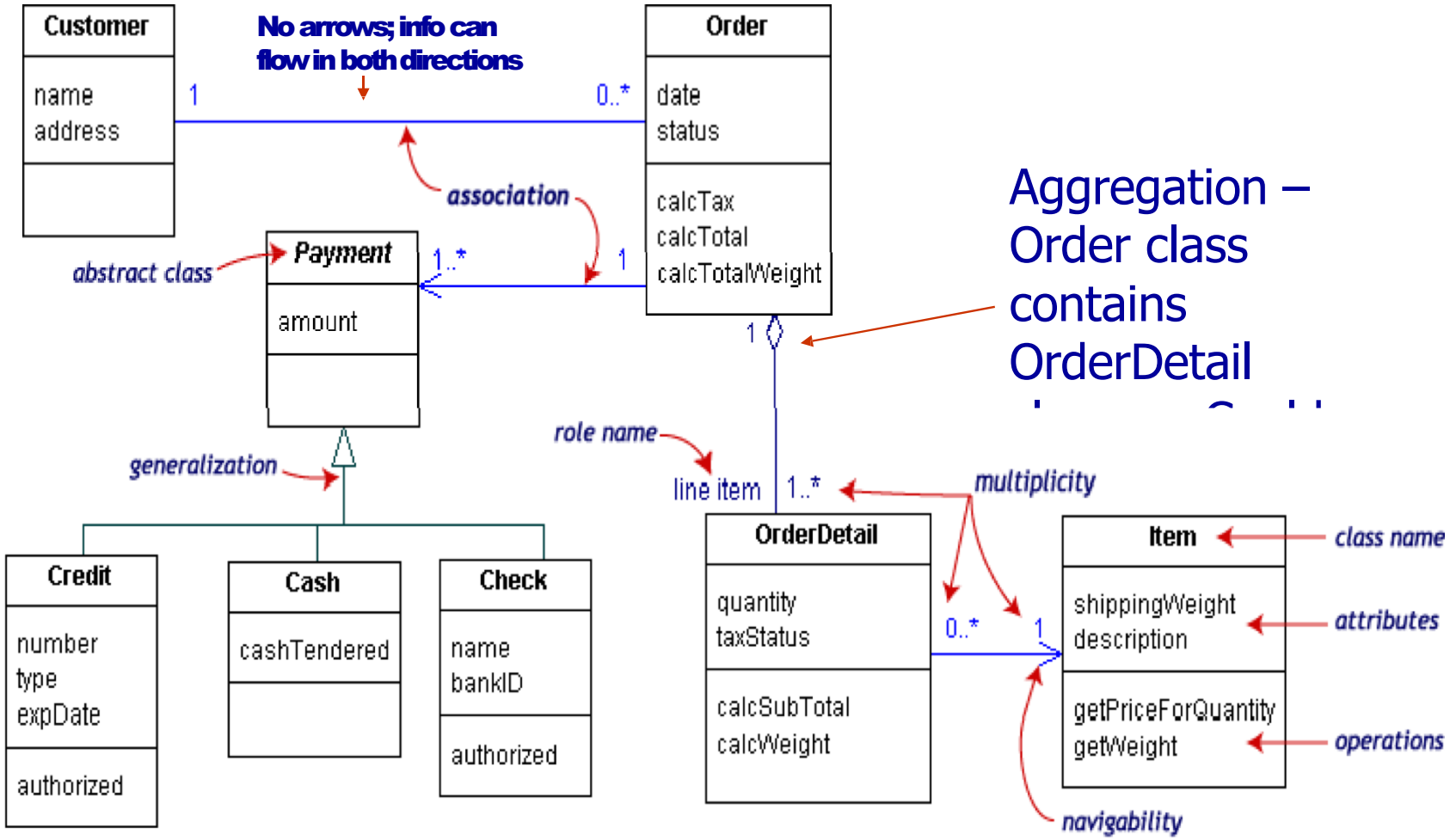


If the movie theatre goes away

so does the box office => composition

but movies may still exist => aggregation

Class diagram example



In Code

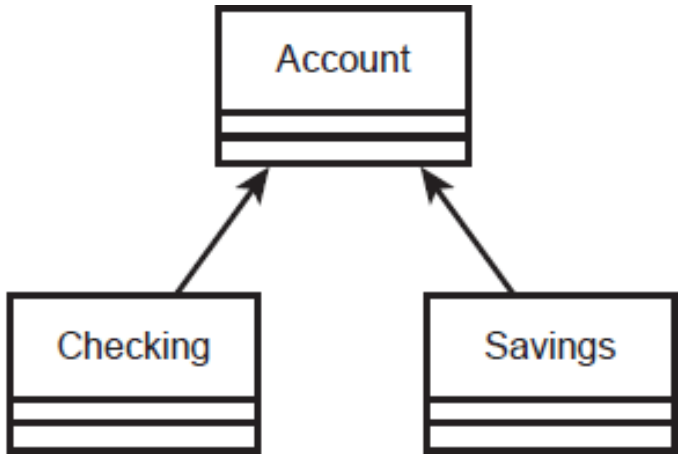
```
class Tenant
{
public:

  Tenant ( );

 ~Tenant ( );

   AptBld * m_livesat;

  void setLivesAt ( AptBld * new_var );

  AptBld * getLivesAt ( );


  AptBld GetApt ( )
  {
  }

  void SetApt (AptBld A1 )
  {
  }

};
```

```
class AptBld
{
public:

  AptBld ( );

  ~AptBld ( );

  vector<Tenant*> m_housesVector;

  void addHouses ( Tenant * add_object );

  void removeHouses ( Tenant * remove_object );

  vector<Tenant *> getHousesList ( );

  Tenant GetTenant ( )
  {
  }

  void SetTenant (Tenant R1 )
  {
  }

};
```

Class Diagram (Associations: N:N)

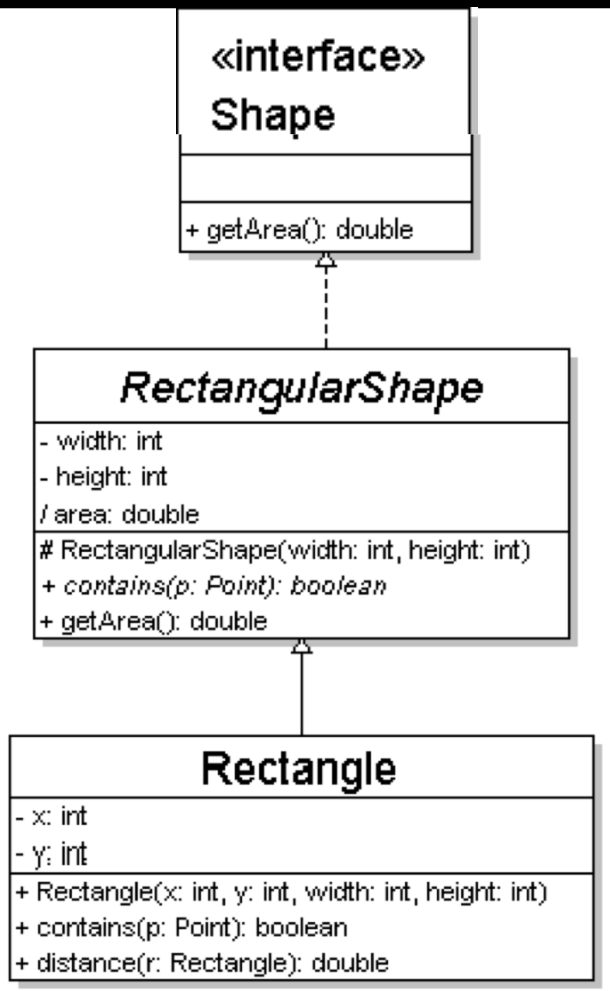
Class Diagram (Generalization “is a”)

A generalization class diagram.



Generalization relationships

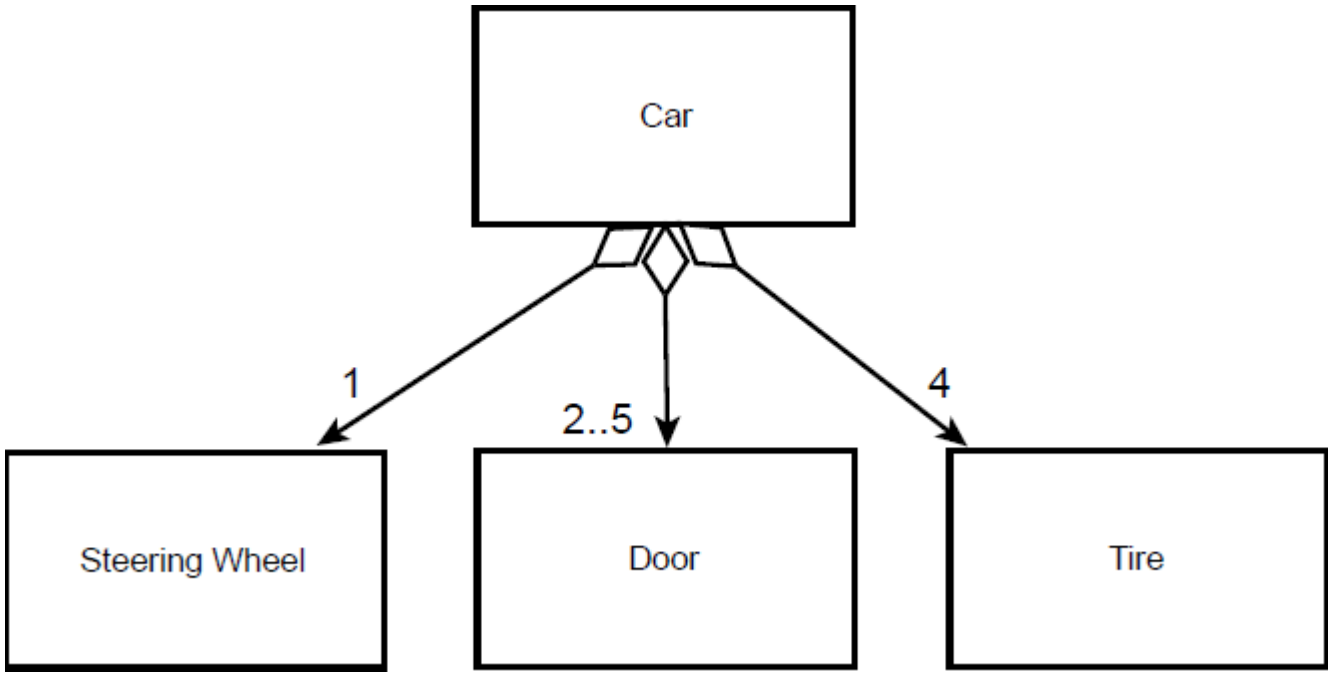
- generalization (inheritance) relationships
  - hierarchies drawn top-down with arrows pointing upward to parent
  - line/arrow styles differ, based on whether parent is a(n):
    - class:
      solid line, black arrow
    - abstract class:
      solid line, white arrow
    - interface:
      dashed line, white arrow
  - we often don't draw trivial / obvious generalization relationships, such as drawing the Object class as a parent



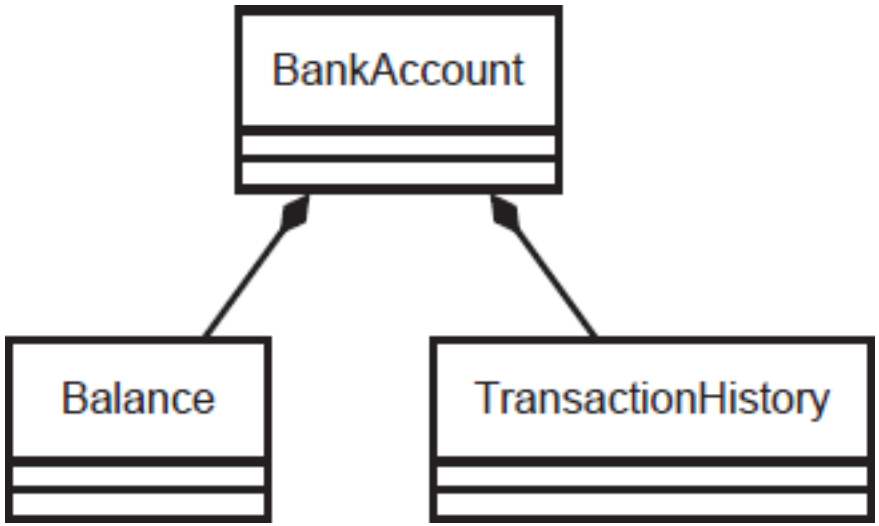
In Code

```
class Current_Account : public Bank_Account
{
public:

};

class Savings_Account : public Bank_Account
{
public:

};
```

Class Diagram (Aggregation “has a”)



Class Diagram (Composition **stronger** "has a")



In Code

```
class Bnk_Accnt
{
public:

  Bnk_Accnt ( );
  ~Bnk_Accnt ( );

  TransacHist GetTransacHist ( )
  {
  }


  void SetTransHist (TransacHist T1 )
  {
  }


  Balance GetBalan ( )
  {
  }


  void SetBalan (Balance B1 )
  {
  }

};
```

```
class Balance
{
public:

  Balance ( );
 ~Balance ( );

  void SetBnkAcc (Bnk_Accnt Bnk1 )
  {
  }

  Bnk_Accnt GetBnkAcc ( )
  {
  }
};
```

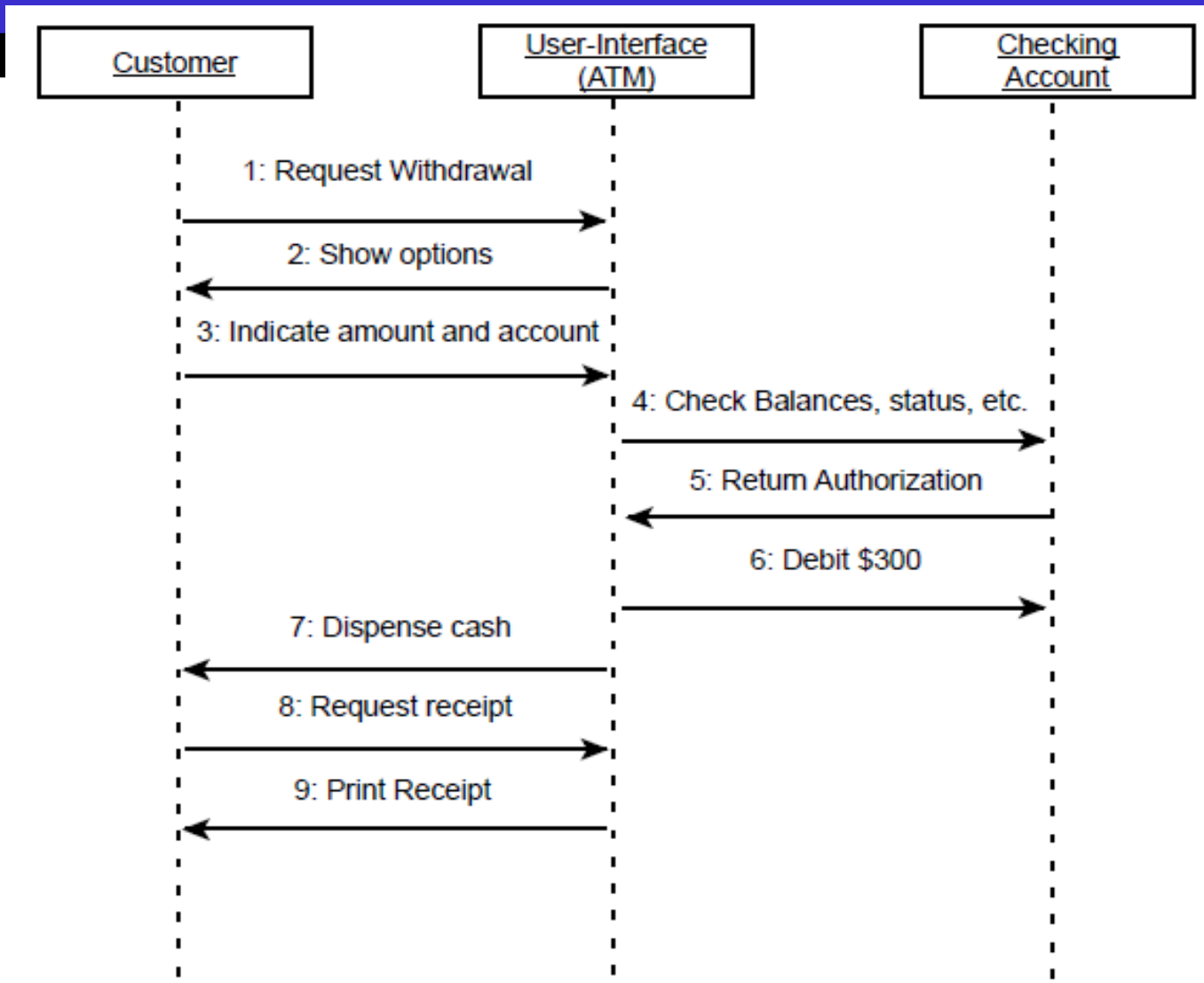
And similarly for class TrasactionHistory

SEQUENCE DIAGRAM (Dynamic)

- GOAL
  - Sequence Diagrams show the message exchange (i.e. method calls) between several Objects in a specific time-delimited situation
  - Sequence Diagrams put special emphasis in the order and the times in which the messages to the objects are sent

- KEY FEATURES
  - Objects are represented through vertical dashed lines, with the name of the Object on the top
  - The time axis is also vertical, increasing downwards, so that messages are sent from one Object to another in the form of arrows with the operation and parameters name
  - Messages can be either synchronous where control is passed to the called object until that method has finished running, or asynchronous where control is passed back directly to the calling object

SEQUENCE DIAGRAM (Example)



UML TOOLS (Make our job easy)

- A UML tool or UML modeling tool is a software application that supports some or all of the notation and semantics associated with the UML

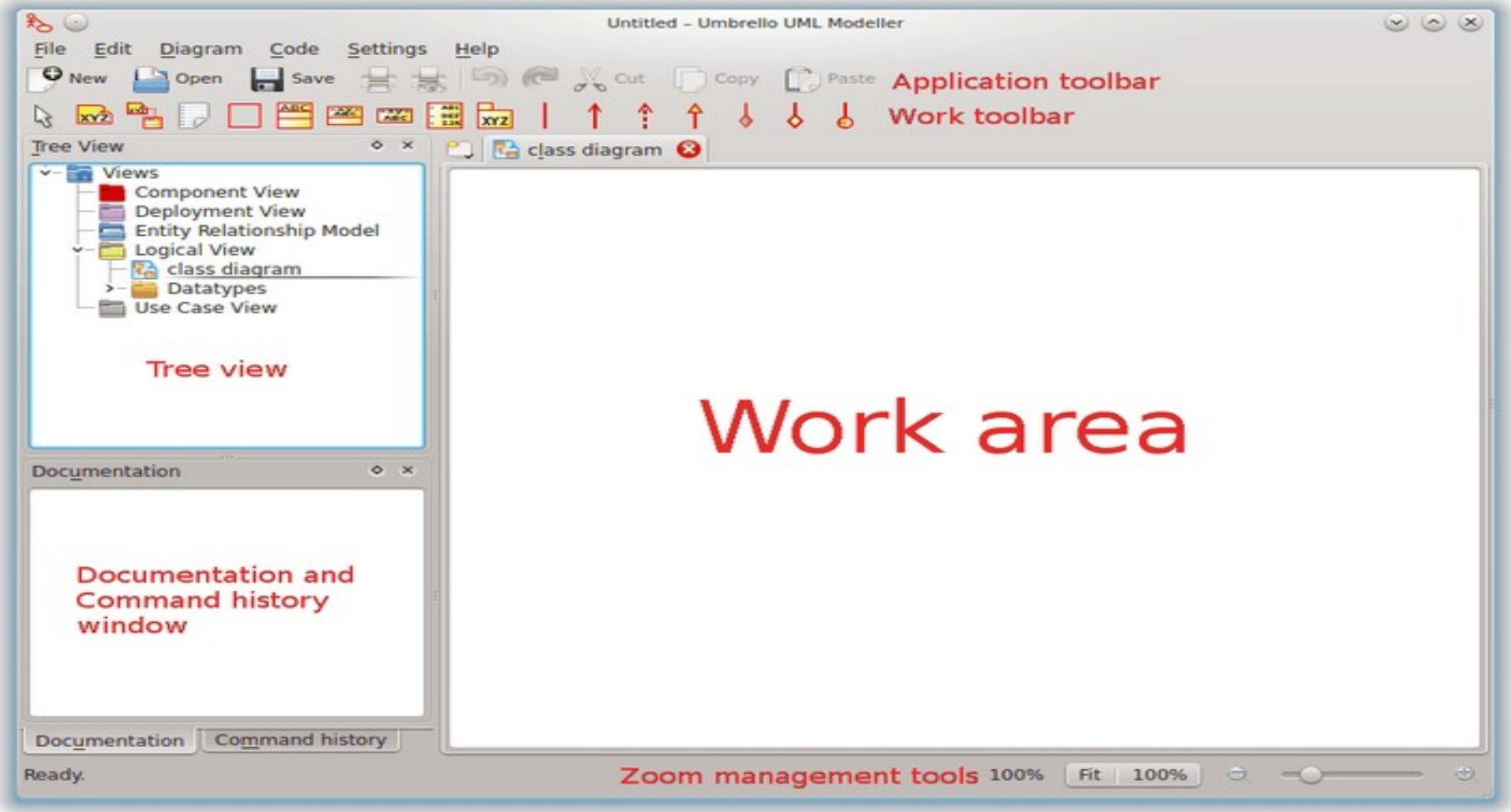
- There are many tools available for Modeling in UML like Rational Rose, TogetherSoft, and ArgoUML (http://plg.uwaterloo.ca/~migod/uml.html). We would be using an open-source tool Umbrello UML Modeller (http://umbrello.kde.org/), which has a nice integration with C++ and Java.

- UML tools support many functionalities like diagramming, code generation, and reverse engineering.

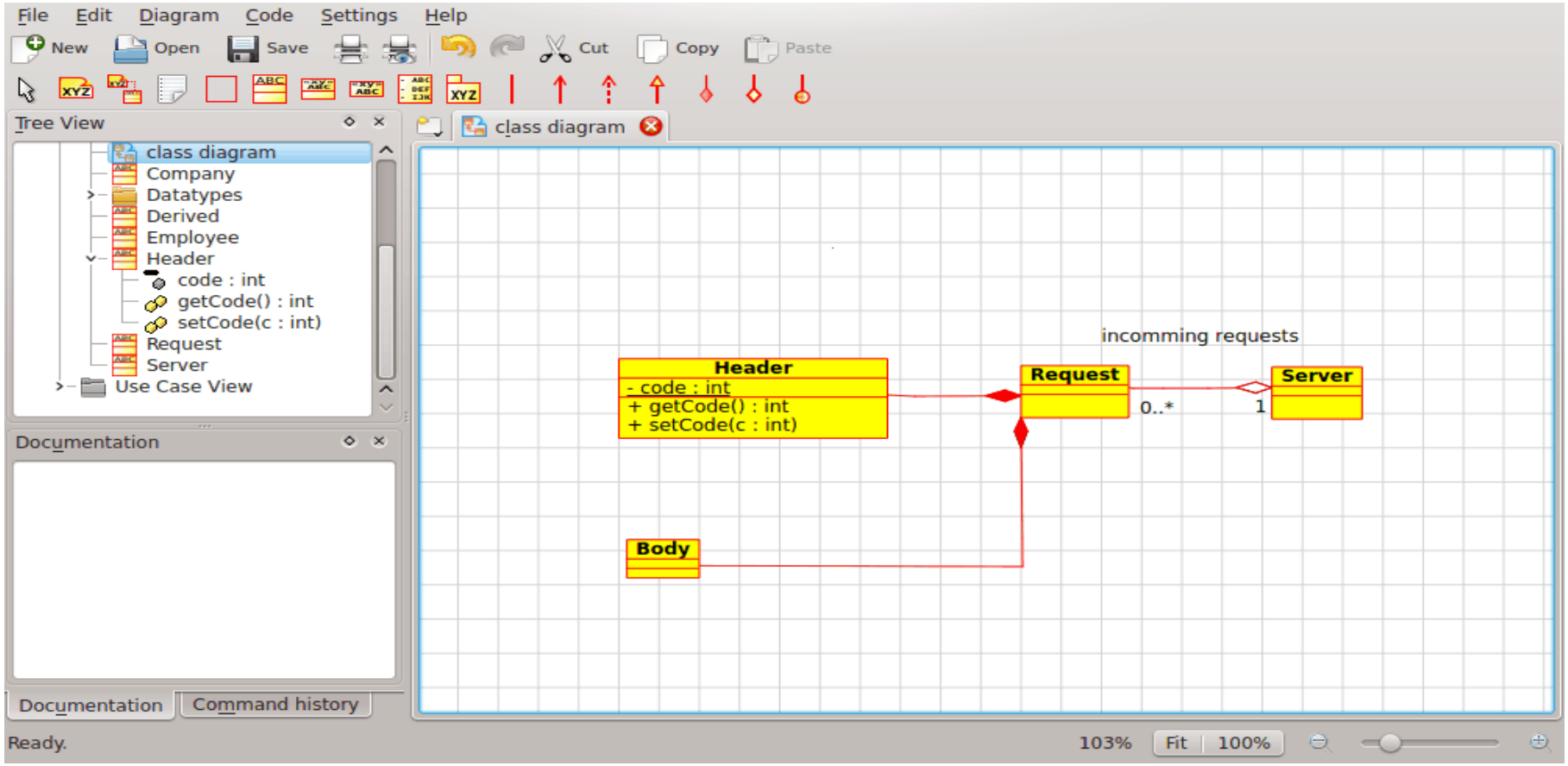
Installing Umbrello

- On Linux, install using sudo apt-get install umbrello
- On Windows, one could take the following steps to install umbrello: http://techbase.kde.org/Projects/KDE_on_Windows/Installation#KDE_Installer_for_Windows (Please search for "umbrella" on the package selection form.)

Umbrello UML Modeller UI



CLASS DIAGRAM ILLUSTRATION



Creating, Loading and Saving Models Pretty Easy!

- The first thing you need to start doing something useful with Umbrello UML Modeller is to create a model to work on. When you start Umbrello UML Modeller it always loads the last used model or creates a new, empty model (depending on your preferences set in the configuration dialog).

- **New Model**

  If at any time you need to create a new model you can do this by selecting the New entry from the File menu, or by clicking on the New icon from the application toolbar. If you are currently working on a model which has been modified Umbrello UML Modeller will ask you if it should save your changes before loading the new model.

Creating, Loading and Saving Models

- **Save Model**
  - You can save your model at any time by selecting the option Save from the File Menu or by clicking on the Save button from the application toolbar. If you need to save your model under a different name you can use the option Save As from the File Menu.

- **Load Model**
  - For loading an already existing model you may select the option Open from the File Menu or click on the Open icon from the application toolbar.

Editing Models

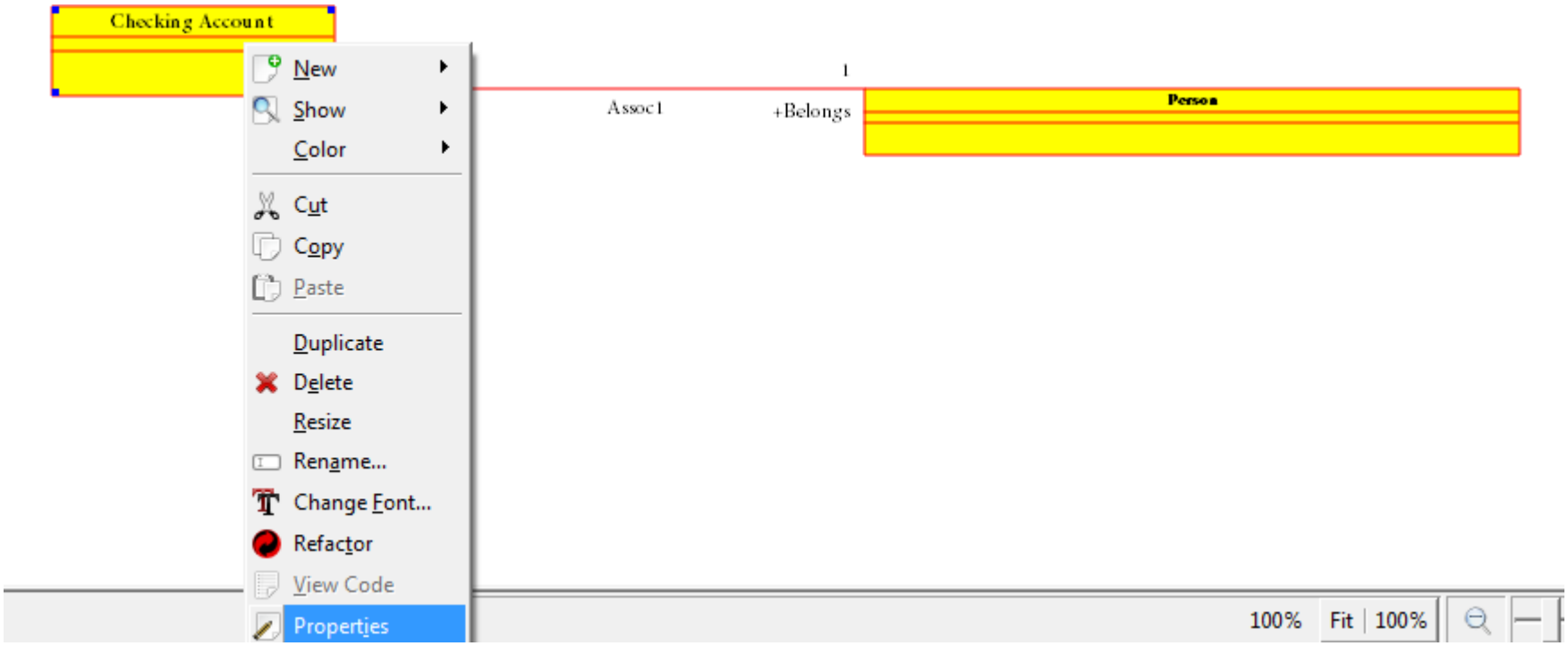
- In Umbrello UML Modeller, there are basically two ways for editing the elements in your model.
  - Edit model elements directly through the Tree View
  - Edit model elements through a Diagram
- Using the context menu of the different items in the Tree View you are able to add, remove, and modify almost all the elements in your model. Right clicking on the folders in the Tree View will give you options for creating the different types of diagrams as well as, depending on whether the folder is a *Use Case View* or a *Logical View*, Actors, Use Cases, Classes, etc.
- Once you have added elements to your model you can also edit an element by accessing its properties dialog, which you find by selecting the option *Properties* from the context menu shown when right clicking on the items in the Tree View.
- You can also edit your model by creating or modifying elements through diagrams. More details on how to do this are given in the following sections.

Editing Diagrams: Edit Elements

- **Editing Elements**

  To edit the properties of an object, select ***Properties*** from its context menu (right mouse button click). Each element has a dialog consisting of several pages where you can configure the options corresponding to that element.

Example


Checking Account
New
Show
Color
Cut
Copy
Paste
Duplicate
Delete
Resize
Rename...
Change Font...
Refactor
View Code
Properties
1
Assoc1
+Belongs
Person
100%
Fit
100%

Code Generation

- Umbrello UML Modeller can generate source code for various programming languages based on your UML Model to help you get started with the implementation of your project. The code generated consists of the class declarations, with their methods and attributes so you can “fill in the blanks” by providing the functionality of your classes' operations.

- Umbrello UML Modeller 2 comes with code generation support for ActionScript, Ada, C++, C#, D, IDL, Java™, JavaScript, MySQL and Pascal.

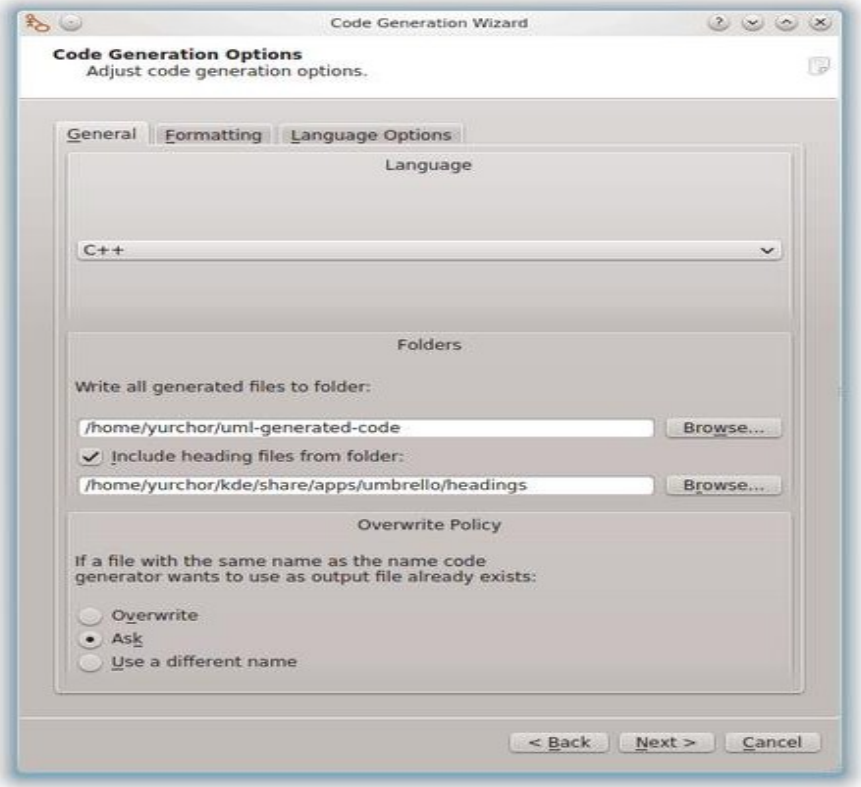
Code Generation

- In order to generate code with Umbrello UML Modeller, you first need to create or load a Model containing at least one class. When you are ready to start writing some code, select the ***Code Generation Wizard*** entry from the ***Code*** menu to start a wizard which will guide you trough the code generation process.

- Steps:
  1. The first step is to select the classes for which you want to generate source code. By default all the classes of your model are selected, and you can remove the ones for which you do not want to generate code by moving them to the left-hand side list
  2. The next step of the wizard allows you to modify the parameters the Code Generator uses while writing your code.

Code Generation

The following options are available:



Options for the Code Generation in Umbrello UML Modeller

3. The third and last step of the wizard shows the status of the Code Generation process. You need only to click on the Generate button to get your classes written for you.

References

- There is plenty of information on UML notations and tools on the web. Main references used for this lecture:
  - C++ Unleashed by Jesse Liberty http://portal.aauj.edu/portal_resources/downloads/uml/programming_object_oriented_analysis_c_plus_plus_unl.pdf
  - http://umbrello.kde.org/
  - http://docs.kde.org/stable/en/kdesdk/umbrello/index.html

Questions/Comments