

Let's build a house!

- ▶ Requirements of Owner
- ▶ Architect makes Plan
- ▶ Engineers make stability calculations
- ▶ Earthquake zone  $\Rightarrow$  special requirements
- ▶ Plan Review and Revision
- ▶ Civil Engineer Approval of Plan
- ▶ Negotiation with Building Company
- ▶ Building
- ▶ Approving and Moving In

Let's build software!

- ▶ Customer requirements
  - ▶ Let's code a prototype
  - ▶ Let's also do some testing
  - ▶ Great, we can show the customer the prototype!
  - ▶ This is nearly what the customer wanted, ok, we change it a bit
  - ▶ Customer is much happier, let's go productive
  - ▶ We are already behind schedule, let's just improve the prototype
  - ▶ Customer's accounting department has new requirements
  - ▶ Let's hack a solution for them
- one year passes
- ▶ Bug: data from this and the last year is mixed
  - ▶ Programmer who hacked the code has left to another company
  - ▶ ...???

# Software

- ▶ Is complex
- ▶ Can be dangerous
  - ▶ Car control
  - ▶ Traffic light control
  - ▶ Power plants
- ▶ Bugs can be very expensive
  - ▶ Satellite has crashed because of miles/km mistake
  - ▶ Millions of users must change passwords because of security bug
- ▶ Satisfying new requirements can be difficult

- ▶ **YOU** will create/manage software
  - ▶ that is complex
  - ▶ that can be dangerous, and
  - ▶ where bugs can become very expensive.
- ▶ Your customers' requirements will **always** evolve over time.

Why?

- ▶ Few customers have exact documentation of their business processes.
- ▶ Processes and requirements are known only to directly involved people.
- ▶ Software is exact (and stupid!)  $\Rightarrow$  Software can do exactly what we make it for. (hopefully)

# Couse Overview

- ▶ Diagrams for Planning and Documenting Software
  - ▶ Unified Modeling Language (UML)
  - ▶ Description and Documentation of Software
  - ▶ Connects "Coders" with Management with Customers
  - ▶ Partially understandable by non-experts (intuitively)
  - ▶ Precise and clear for experts (with training)
- ⇒ essential part of Software Requirements Specification (SRS)
- ▶ Typical Problems in Software Design and how to solve them
  - ▶ Design Patterns = reusable solutions to common situations
  - ▶ Structure of software
  - ▶ How to realize patterns in software

# Sources and Resources

## Acknowledgements:

- ▶ Some images from Wikipedia
- ▶ Some slides from Bob Tarr
- ▶ Some Slides from Timothy Lethbridge and Robert Laganière

## Books:

- ▶ Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software (1995).
- ▶ Martin Fowler: UML Distilled: A Brief Guide to the Standard Object Modeling Language (1997).
- ▶ Many more ...

# Software Projects

- ▶ A project is an effort to reach a **goal** within a **limited time**.
- ▶ Success within the time always requires **planning**.
- ▶ Projects are usually **collaborative** efforts.
- ▶ The concrete goals might not be completely clear/become clear during the project.
- ▶ A project usually exists for a limited amount of time.

# Usual Activities in a Software Project

- ▶ Requirements Analysis

"What do we need to build actually?"

- ▶ Software Design

"What is the most clever way to build this?"

- ▶ Coding

"Transforming our design into code."

- ▶ Testing

"Does it work?"

Every step can mean "restart from the very beginning".



# Requirements Analysis

"What do we want to build actually?"

- ▶ Manager of customer has a problem or a vision.
  - ▶ Problems come from some department in the company.
  - ▶ Visions come from ideas and might not be realizable.
- ▶ Requirements analysis mainly consists of **communication**.
  - ▶ People have different knowledge and different assumptions.
  - ▶ Communication is often unclear.
  - ▶ Important things are often assumed and not communicated.

# Requirements Analysis

"What do we want to build actually?"

- ▶ Manager of customer has a problem or a vision.
  - ▶ Problems come from some department in the company.
  - ▶ Visions come from ideas and might not be realizable.
- ▶ Requirements analysis mainly consists of **communication**.
  - ▶ People have different knowledge and different assumptions.
  - ▶ Communication is often unclear.
  - ▶ Important things are often assumed and not communicated.
  - ▶ People are afraid to say "Sorry, I do not understand this."  
(This fear can become very expensive.)  
((Hint: your BSc thesis will be a project.))

# Artifacts

The result of a project is a collection of **artifacts**.

- ▶ Source Code
- ▶ Documentation
- ▶ Tests
- ▶ Website
- ▶ Installation
- ▶ Installer package
- ▶ Scientific paper
- ▶ ...

Sometimes the result can be an activity

- ▶ Presentation
- ▶ Concert
- ▶ Show

# Software Design - Clever

"What is the most clever way to build this?"

What is clever?

- ▶ Depends on Requirements
- ▶ Speed
- ▶ Flexibility
- ▶ Operating vs Building Cost

**Tradeoffs!** (no unique best solution)

# Software Design - Reuse

Can we **reuse** existing artifacts?

- ▶ Do these fit our purpose?
- ▶ Do we have the knowledge to handle them?
- ▶ Do we have the premission (license) to use them?

# Software Design - Target Environment

What is our target environment?

Some possibilities:

- ▶ Closed group of experts in one of customer's departments.  
⇒ we can use a difficult setup process
- ▶ Public market
- ▶ Multi-Platform
  - ▶ PC/Mac/Linux
  - ▶ Android/iOS/Windows Phone/Blackberry
  - ▶ Xbox, Playstation, Wii, ...?
- ▶ Mobile users with phone/tablet  
⇒ different visual/interaction design
- ▶ Single installation in one computer  
⇒ we can fully specify HW and SW, including operating system
- ▶ Embedded system in cars/devices  
⇒ controlled HW/SW but many other requirements (safety, upgrade safety, limited resources, often realtime requirements)

# Software Design - Components

How can we **decompose** our artifacts into components?

- ▶ Smaller components can be **developed and tested** easier. Writing one function that "does everything" will go very wrong!
- ▶ **Separating concerns** is a good strategy.  
Example concerns:
  - ▶ security
  - ▶ numerical computations (statistics, ...)
  - ▶ user interface
  - ▶ communication
  - ▶ data storage

Not all concerns can be completely separated into components (e.g., security and communication)  $\Rightarrow$  tradeoffs!

# Software Design - Interactions

How will our components **interact**?

- ▶ Some might be parts of other components  
e.g., "menu item" part of "menu bar" part of "window"
- ▶ Some might refer to other components  
e.g., "menu item" knows the "view" so that it can reset the zoom
- ▶ They might create other components.  
e.g., "open file" dialog creates a new "file loader" object  
e.g., "file loader" object creates "view", "document", ...



# Coding

Transforming our design into code.

- ▶ We make our ideas formal.
- ▶ So formal that **even a computer can "understand"** them.
- ▶ Computers are stupid ...
  - ▶ therefore we must cover **every possibility**
  - ▶ therefore we must **define all decisions**

Often this will show us that our design is not complete.

The design might be incomplete because our analysis is not complete.

"Back to start?"

# Testing

"Does it work?"

- ▶ Programmer claims it works.
- ▶ Does it work as the programmers expect?

It might actually work!

This is only the start.

- ▶ Does it work as the management expect?

Yes, if communication is really good.

- ▶ Does it work as the customer's management expect?

Yes, if communication with customer is really good.

- ▶ Does it work as the customer's employees/users expect?

Yes, if ...

Usually: No!

# Testing - a lost cause?

The only lost cause is, if there is no Testing!

- ▶ Testing within the company improves efficiency and quality.
- ▶ Testing can be done on parts of the product.  
"Unit testing": test small units
- ▶ Testing can be automated
  - ▶ Every new commit in the source code runs all tests
  - ▶ Mistakes are found fast and can be repaired fast
- ▶ How much testing do we need?
  - ▶ Guideline: LOC for unit tests = LOC for software product
  - ▶ Additional system tests/system part tests/integration tests

# SMART Projects

- ▶ **Specific**  
Have a clear specific goal to achieve.
- ▶ **Measurable**  
Be able to measure the progress towards that goal.
- ▶ **Achievable**  
Have an idea how to achieve the goal.
- ▶ **Relevant**  
Perform projects that are worthwhile.  
... worthwhile to you.  
... worthwhile to people that you need to contribute/help.
- ▶ **Time-bound**  
Use deadlines.  
There is always something coming up — a deadline motivates to overcome these problems.

SMART projects are **more fun** and **more successful**.

# Parkinson's law

*Work expands so as to fill the time available for its completion*



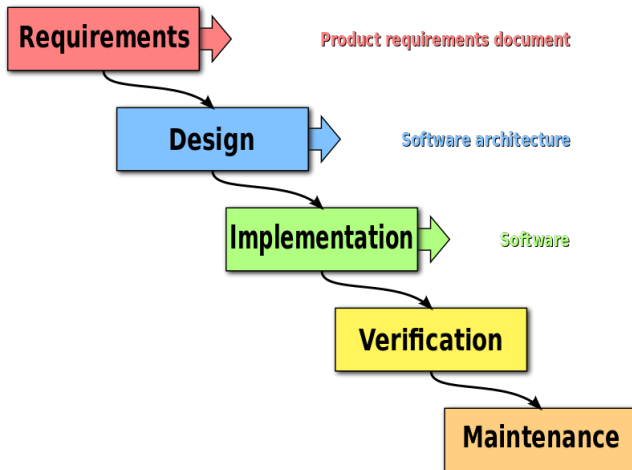
- ⇒ shorter and more deadlines can multiply productivity
- This also works for studying - homeworks, learning, thesis writing.
- ⇒ split your work into smaller pieces and attack them one by one!

(Picture and quote from [www.quora.com](http://www.quora.com))

# Software Development Process Models

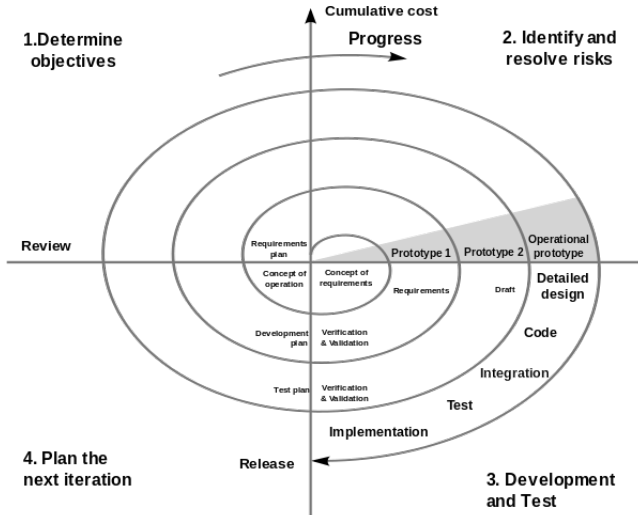
- ▶ There are many ways of managing/designing Software
- ▶ Waterfall Model
- ▶ Spiral Model
- ▶ Iterative Development
- ▶ Agile Development
  - ▶ Scrum
  - ▶ Extreme Programming (XP)

# Waterfall Model



- Phase starts after previous phase finished.

## Spiral Model

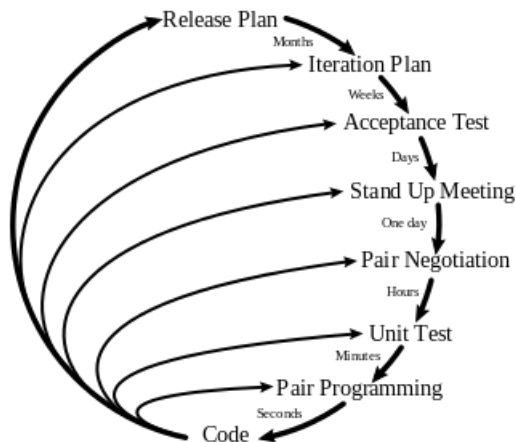


- ▶ Phases start short and rough.
- ▶ Phases become longer and prototypes become more detailed.



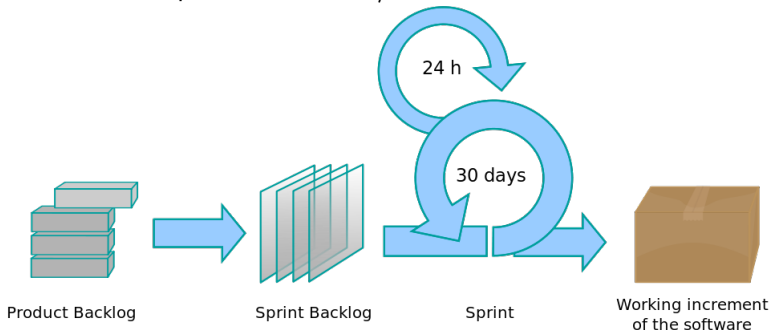
# Extreme Programming

- ▶ Multiple short development cycles - changes are inherent to software-development.
- ▶ Coding, Testing, Listening (to customers), **Designing** (because coding, testing, listening is not sufficient).
- ▶ Pair programming.



# Scrum

- ▶ Roles: Product Owner, Scrum Master, Developers (3-9)
- ▶ Self-organizing development team.
- ▶ **Sprint**: one iteration of development (timeboxed 7-30 days)  
The goal of each sprint is a 'potentially shippable item'.
- ▶ 15min meeting each morning:
  - ▶ What was done yesterday, what needs to be done today?
  - ▶ Which problems did arise/are unsolved?



# Software Requirements Specification (SRS)

- ▶ Requirements Specification (from customer)  
"What do we need in the software?"
- ▶ Functional Specification (from software company)  
"We offer to realize what you need the following way!"
- ▶ Ideal World: several bidders with different functional specifications, customer selects the objectively best offer.
- ▶ Reality is different. (Especially in Turkey.)
- ▶ Without specification a software project **will go very wrong**.

# Questions about Software Development Processes

Question: → Why are there so many Software Development Process models? (Are they appropriate for different kinds of software/customer?) ←

Question: → Who is responsible for Software Design in agile methods (Scrum, Extreme Programming)? ←

Question: → Explain how much we care about Software Design in the four activities (Requirements Analysis/Software Design/Coding/Testing) ←

Question: → If SMART is so good, why do many people not use it? What are problems with SMART methodology? ←

## Example

*We have a game field where a piece composed of squares falls down from the top of the field until it hits another piece or the bottom. Then the piece stops and stays there and the next piece comes from above. The pieces fall slowly, the player can rotate them by 90 degrees and move them left and right with the keyboard. If there is one row completely filled by pieces, this row is cut out of the playing area and all rows above are moved down by one row. The stones fall faster and faster the longer the game runs. The player loses if a new piece cannot enter the screen because it is filled with pieces.*

**Question: → Which information is missing in the specification? ←**

## Example (Something Like Tetris)

*We have a game field where a piece composed of squares falls down from the top of the field until it hits another piece or the bottom. Then the piece stops and stays there and the next piece comes from above. The pieces fall slowly, the player can rotate them by 90 degrees and move them left and right with the keyboard. If there is one row completely filled by pieces, this row is cut out of the playing area and all rows above are moved down by one row. The stones fall faster and faster the longer the game runs. The player loses if a new piece cannot enter the screen because it is filled with pieces.*

**Question:** → Which information is missing in the specification? ←

**Question:** → Imagine you do not know Tetris: which wrong assumptions could you make? ←

# UML - Unified Modeling Language

- ▶ A tool for describing Software Designs by means of Diagrams.
- ▶ Integrated into software tools and even into software development processes.
- ▶ XML standard for storing/exchanging diagrams.
- ▶ Important Issues / Warnings:
  - ▶ UML is **not rigorously defined**  
Some aspects can be interpreted in different ways.
  - ▶ UML is **not rigorously used** by humans  
Even clearly defined aspects will be interpreted in different ways by different people.
  - ▶ UML diagrams **can be incomplete**  
It is not required to put everything into a diagram.  
(Diagrams can be much clearer if they omit some information.)

# Why do we care about UML?

What is the good side of UML?

- ▶ UML diagrams tell a lot about software.
- ▶ UML diagrams make communication between humans more efficient.  
(Common ground for discussions becomes much larger.)
- ▶ UML diagrams can help to talk about software on several **levels of abstraction**.  
(Component, Package, Class, Method)
- ▶ UML diagrams can make us **understand software** easier.



## More Warnings

- ▶ Human Communication is still required!
- ▶ Unclear things:
  - ▶ Lookup in the standard/UML books
    - ⇒ might become clear
    - (⇒ might not be the intended usage)
  - ▶ Ask the author(s) of the diagram
    - ⇒ safer method

## 2.1 What is Object Orientation?

### Procedural paradigm:

- Software is organized around the notion of *procedures*
- *Procedural abstraction*
  - Works as long as the data is simple
- *Adding data abstractions*
  - Groups together the pieces of data that describe some entity
  - Helps reduce the system's complexity.
    - Such as *Records* and *structures*

### Object oriented paradigm:

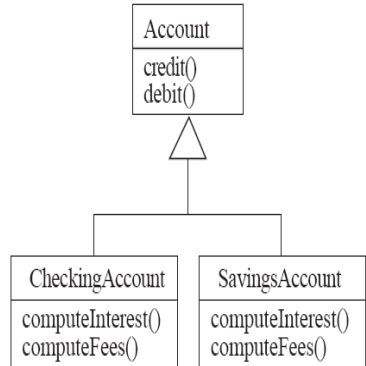
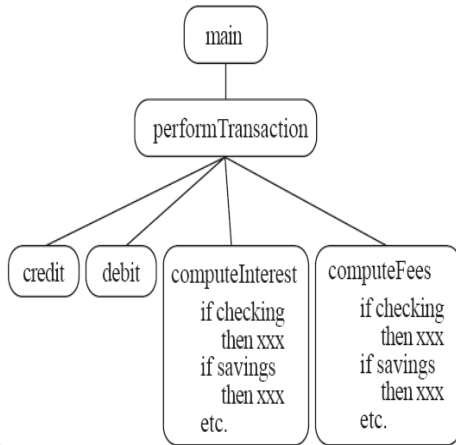
- Organizing procedural abstractions in the context of data abstractions

# Object Oriented paradigm

**An approach to the solution of problems in which all computations are performed in the context of objects.**

- The objects are instances of classes, which:
  - are data abstractions
  - contain procedural abstractions that operate on the objects
- A running program can be seen as a collection of objects collaborating to perform a given task

# A View of the Two paradigms



See in Umple

## 2.2 Classes and Objects

### Object

- A chunk of structured data in a running software system
- Has *properties*
  - Represent its state
- Has *behaviour*
  - How it acts and reacts
  - May simulate the behaviour of an object in the real world



# Objects

<u>Greg:</u>
dateOfBirth="1970/01/01" address="75 Object Dr."

<u>Jane:</u>
dateOfBirth="1955/02/02" address="99 UML St." position="Manager"

<u>Savings account 12876:</u>
balance=1976.32 opened="1999/03/03"

<u>Margaret:</u>
dateOfBirth="1984/03/03" address="150 C++ Rd." position="Teller"

<u>Instant teller 876:</u>
location="Java Valley Cafe"

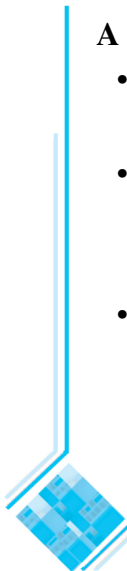
<u>Mortgage account 29865:</u>
balance=198760.00 opened="2003/08/12" property="75 Object Dr."

<u>Transaction 487:</u>
amount=200.00 time="2001/09/01 14:30"

# Classes

## A class:

- A unit of abstraction in an object oriented (OO) program
- Represents similar objects
  - Its *instances*
- A kind of software module
  - Describes its instances' structure (properties)
  - Contains *methods* to implement their behaviour



# Is Something a Class or an Instance?

- Something should be a *class* if it could have instances
- Something should be an *instance* if it is clearly a *single* member of the set defined by a class

## *Film*

- Class; instances are individual films.

## *Reel of Film:*

- Class; instances are physical reels

## *Film reel with serial number SW19876*

- Instance of **ReelOfFilm**

## *Science Fiction*

- Instance of the class **Genre**.

## *Science Fiction Film*

- Class; instances include 'Star Wars'

## *Showing of 'Star Wars' in the Phoenix Cinema at 7 p.m.:*

- Instance of **ShowingOfFilm**



# Naming classes

- Use *capital* letters  
—E.g. BankAccount not bankAccount
- Use *singular* nouns
- Use the right level of generality  
—E.g. Municipality, not City
- Make sure the name has only *one* meaning  
—E.g. 'bus' has several meanings

## 2.3 Instance Variables

**Variables defined inside a class corresponding to data present in each instance**

- Also called *fields* or *member variables*
- Attributes
  - Simple data
  - E.g. name, dateOfBirth
- Associations
  - Relationships to other important classes
  - E.g. supervisor, coursesTaken

# Variables vs. Objects

## **A variable**

- *Refers* to an object
- May refer to different objects at different points in time

**An object can be referred to by several different variables at the same time**

## ***Type* of a variable**

- Determines what classes of objects it may contain



# Class variables

**A *class variable*'s value is *shared* by all instances of a class.**

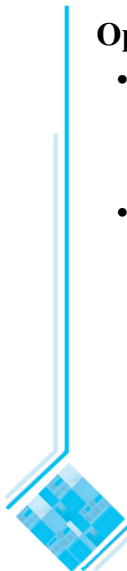
- Also called a *static* variable
- If one instance sets the value of a class variable, then all the other instances see the same changed value.
- Class variables are useful for:
  - Default or 'constant' values (e.g. PI)
  - Lookup tables and similar structures

Caution: *do not over-use class variables*

## 2.4 Methods, Operations and Polymorphism

### Operation

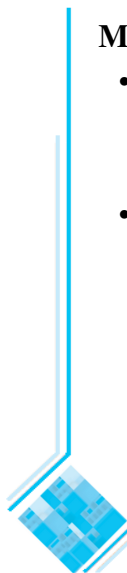
- A higher-level procedural abstraction that specifies a type of behaviour
- Independent of any code which implements that behaviour
  - E.g. calculating area (in general)



# Methods, Operations and Polymorphism

## Method

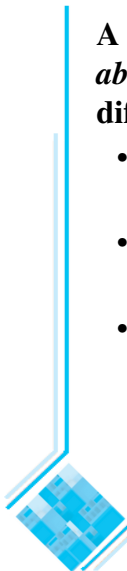
- A procedural abstraction used to implement the behaviour of a class
- Several different classes can have methods with the same name
  - They implement the same abstract operation in ways suitable to each class
  - E.g. calculating area in a rectangle is done differently from in a circle



# Polymorphism

**A property of object oriented software by which an *abstract operation may be performed in different ways in different classes.***

- Requires that there be *multiple methods of the same name*
- The choice of which one to execute depends on the object that is in a variable
- Reduces the need for programmers to code many `if - else` or `switch` statements




## 2.5 Organizing Classes into Inheritance Hierarchies

### Superclasses

- Contain features common to a set of subclasses

### Inheritance hierarchies

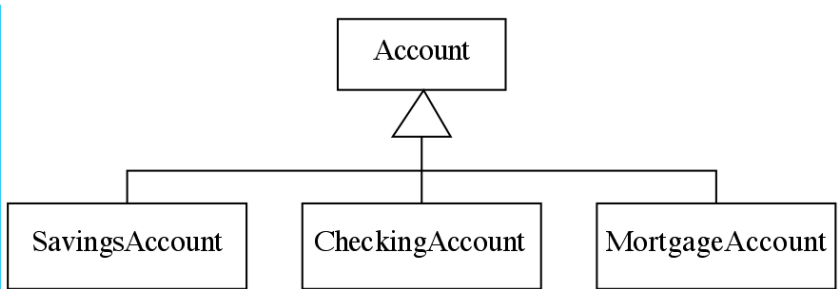
- Show the relationships among superclasses and subclasses
- A triangle shows a *generalization* 

### Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses



# An Example Inheritance Hierarchy



See in Umlpe

## Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

# The Isa Rule

**Always check generalizations to ensure they obey the isa rule**

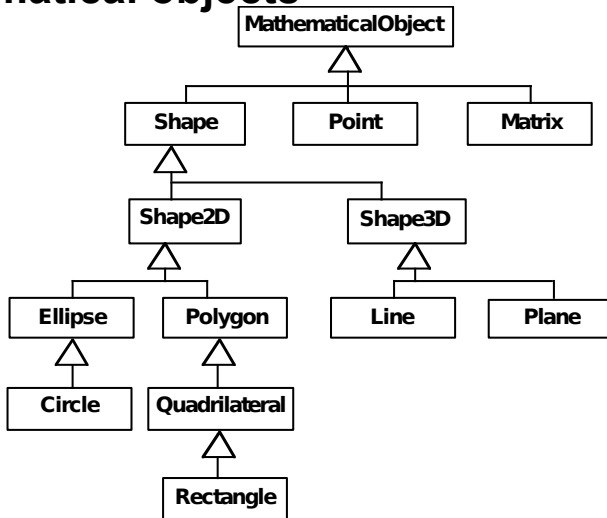
- “A checking account *is an* account”
- “A village *is a* municipality”

**Should ‘Province’ be a subclass of ‘Country’?**

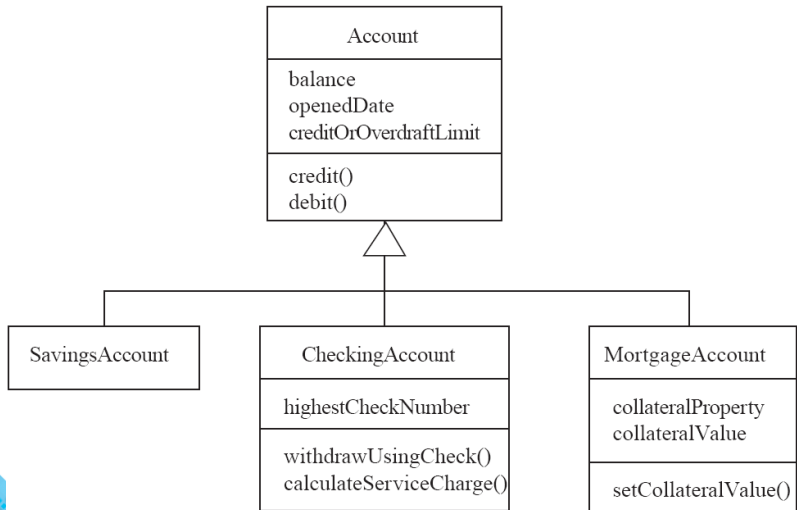
- No, it violates the isa rule
  - “A province *is a* country” is invalid!



# A possible inheritance hierarchy of mathematical objects



# Make Sure all Inherited Features Make Sense in Subclasses



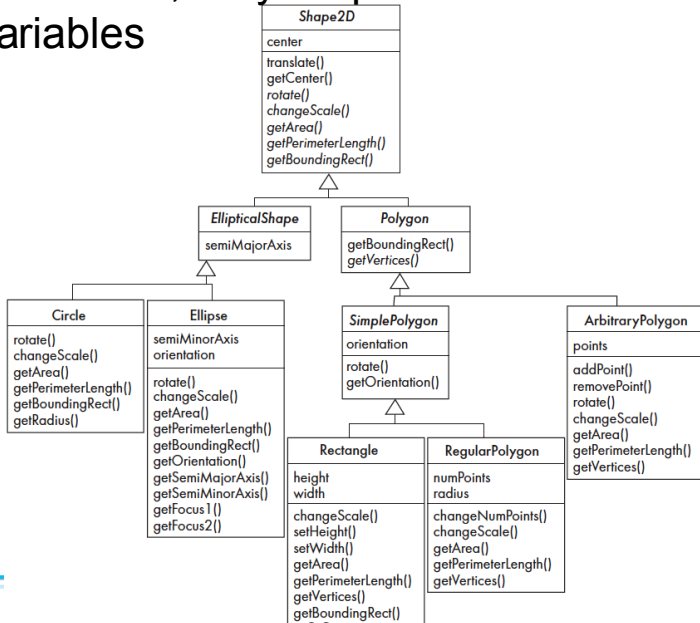
## Example (Something Like Tetris)

*We have a game field where a piece composed of squares falls down from the top of the field until it hits another piece or the bottom. Then the piece stops and stays there and the next piece comes from above. The pieces fall slowly, the player can rotate them by 90 degrees and move them left and right with the keyboard. If there is one row completely filled by pieces, this row is cut out of the playing area and all rows above are moved down by one row. The stones fall faster and faster the longer the game runs. The player loses if a new piece cannot enter the screen because it is filled with pieces.*

**Blackboard:** → **Classes for Something Like Tetris** ←

**Blackboard:** → **Objects for Something Like Tetris** ←

## 2.6 Inheritance, Polymorphism and Variables



# Some Operations in the Shape Example

Original objects  
(showing bounding rectangle)



Rotated objects  
(showing bounding rectangle)



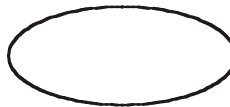
Translated objects  
(showing original)



Scaled objects  
(50%)



Scaled objects  
(150%)



[www.lloseng.com](http://www.lloseng.com)

# Abstract Classes and Methods

**An operation should be declared to exist at the highest class in the hierarchy where it makes sense**

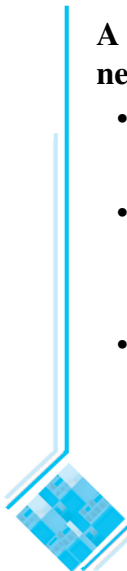
- The *operation* may be *abstract* (lacking implementation) at that level
- If so, the *class* also must be *abstract*
  - No instances can be created
  - The opposite of an abstract class is a *concrete* class
- If a superclass has an abstract operation then its subclasses at some level must have a concrete method for the operation
  - Leaf classes must have or inherit concrete methods for all operations
  - Leaf classes must be concrete



# Overriding

**A method would be inherited, but a subclass contains a new version instead**

- For restriction
  - E.g. `scale(x, y)` would not work in `Circle`
- For extension
  - E.g. `SavingsAccount` might charge an extra fee following every debit
- For optimization
  - E.g. The `getPerimeterLength` method in `Circle` is much simpler than the one in `Ellipse`



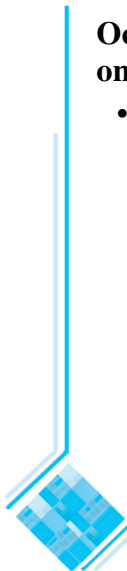
# How a decision is made about which method to run

- 1. If there is a concrete method for the operation in the current class, run that method.**
- 2. Otherwise, check in the immediate superclass to see if there is a method there; if so, run it.**
- 3. Repeat step 2, looking in successively higher superclasses until a concrete method is found and run.**
- 4. If no method is found, then there is an error**
  - In Java and C++ the program would not have compiled

# Dynamic binding

**Occurs when decision about which method to run can only be made at *run time***

- Needed when:
  - A variable is declared to have a superclass as its type, and
  - There is more than one possible polymorphic method that could be run among the type of the variable and its subclasses



## 2.7 Concepts that Define Object Orientation

The following are necessary for a system or language to be OO

- Identity
  - Each object is *distinct* from each other object, and *can be referred to*
  - Two objects are distinct *even if they have the same data*
- Classes
  - The code is organized using classes, each of which describes a set of objects
- Inheritance
  - The mechanism where features in a hierarchy inherit from superclasses to subclasses
- Polymorphism
  - The mechanism by which several methods can have the same name and implement the same abstract operation.

## 5.2 Essentials of UML Class Diagrams

*The main symbols shown on class diagrams are:*

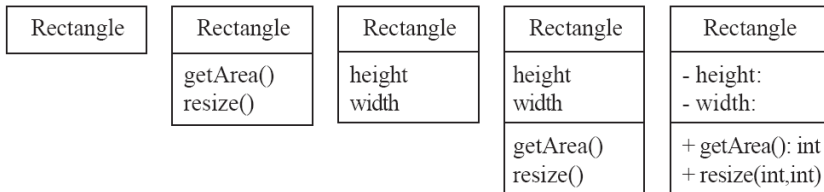
- *Classes*
  - represent the types of data themselves
- *Associations*
  - represent linkages between instances of classes
- *Attributes*
  - are simple data found in classes and their instances
- *Operations*
  - represent the functions performed by the classes and their instances
- *Generalizations*
  - group classes into inheritance hierarchies

# Classes

**A class is simply represented as a box with the name of the class inside**

- The diagram may also show the attributes and operations
- The complete signature of an operation is:

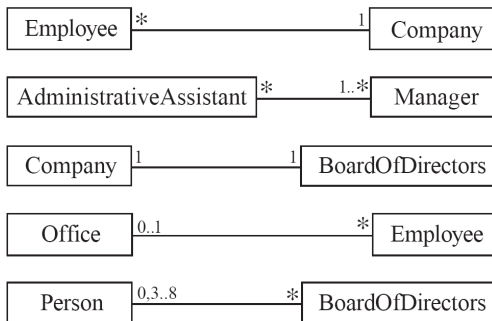
operationName(parameterName: parameterType ...): returnType



## 5.3 Associations and Multiplicity

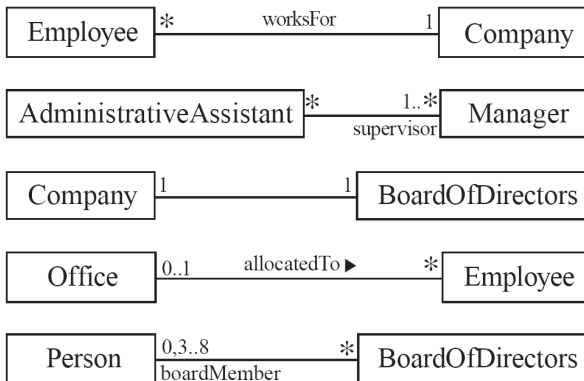
**An *association* is used to show how two classes are related to each other**

- Symbols indicating *multiplicity* are shown at each end of the association



# Labelling associations

- Each association can be labelled, to make explicit the nature of the association

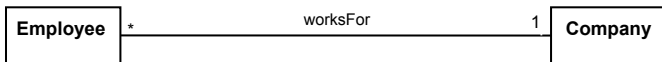




# Analyzing and validating associations

- **Many-to-one**

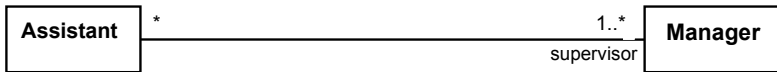
- A company has many employees,
- An employee can only work for one company.
  - This company will not store data about the moonlighting activities of employees!
- A company can have zero employees
  - E.g. a 'shell' company
- It is not possible to be an employee unless you work for a company



# Analyzing and validating associations

- **Many-to-many**

- An assistant can work for many managers
- A manager can have many assistants
- Assistants can work in pools
- Managers can have a group of assistants
- Some managers might have zero assistants.
- Is it possible for an assistant to have, perhaps temporarily, zero managers?



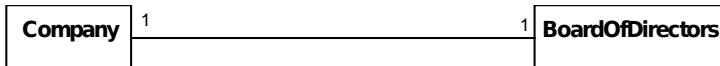
Open in Umpole

[www.lloseng.com](http://www.lloseng.com)

# Analyzing and validating associations

- **One-to-one**

- For each company, there is exactly one board of directors
- A board is the board of only one company
- A company must always have a board
- A board must always be of some company

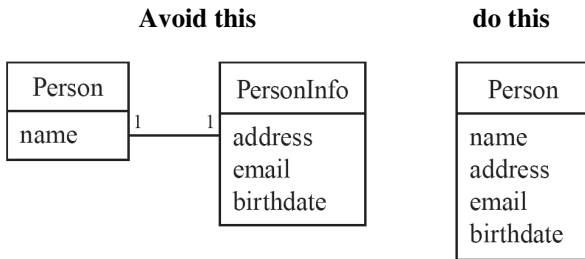


Open in Umple

[www.lloseng.com](http://www.lloseng.com)

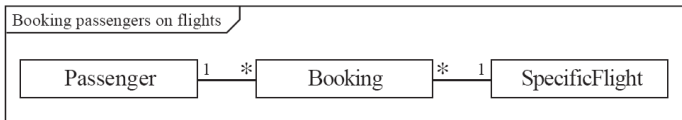
# Analyzing and validating associations

## Avoid unnecessary one-to-one associations



# A more complex example

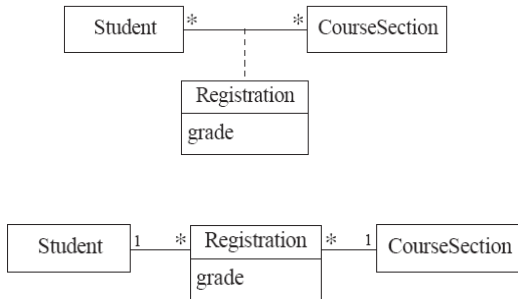
- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could *never* involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking



- The *frame* around this diagram is an optional feature that any UML 2.0 may possess.

# Association classes

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent

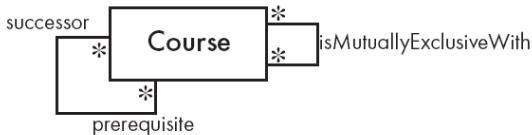


Open in Umpel and extended example

[www.lloseng.com](http://www.lloseng.com)

# Reflexive associations

- It is possible for an association to connect a class to itself



Open in Umpole

[www.lloseng.com](http://www.lloseng.com)

# Directionality in associations

- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end



Open in Umple

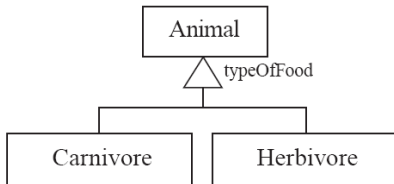
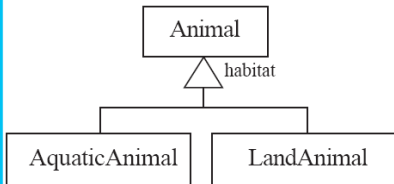
[www.lloseng.com](http://www.lloseng.com)



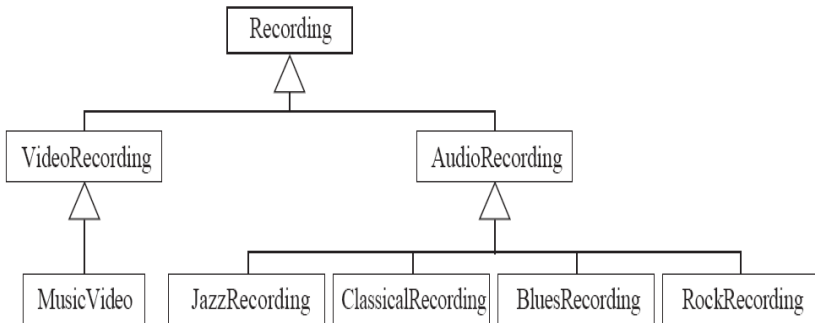
## 5.4 Generalization

### Specializing a superclass into two or more subclasses

- A *generalization set* is a labeled group of generalizations with a common superclass
- The label (sometimes called the *discriminator*) describes the criteria used in the specialization



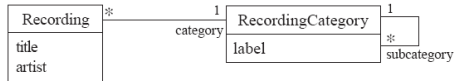
# Avoiding unnecessary generalizations



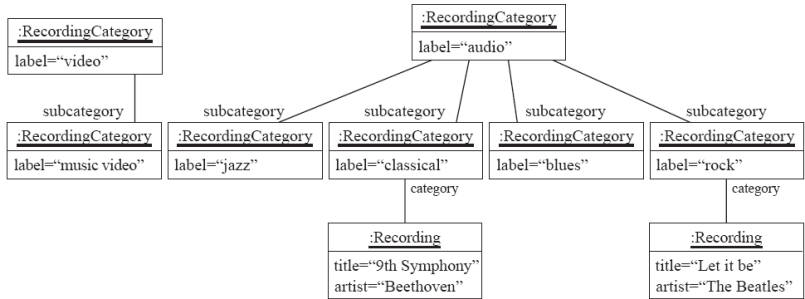
Inappropriate hierarchy of classes, which should be instances

# Avoiding unnecessary generalizations (cont)

Open in Umpole



(a)

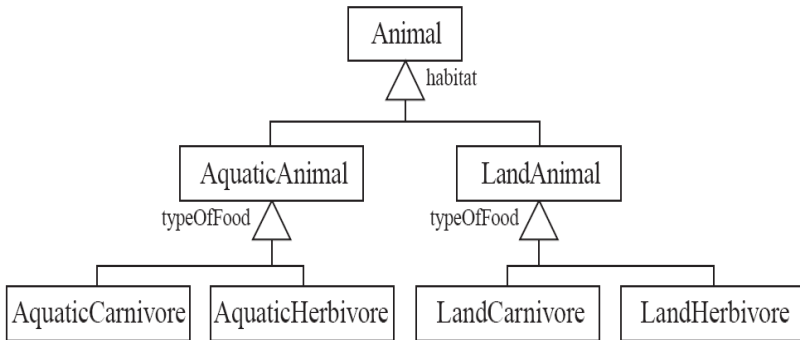


(b)

Improved class diagram, with its corresponding instance diagram

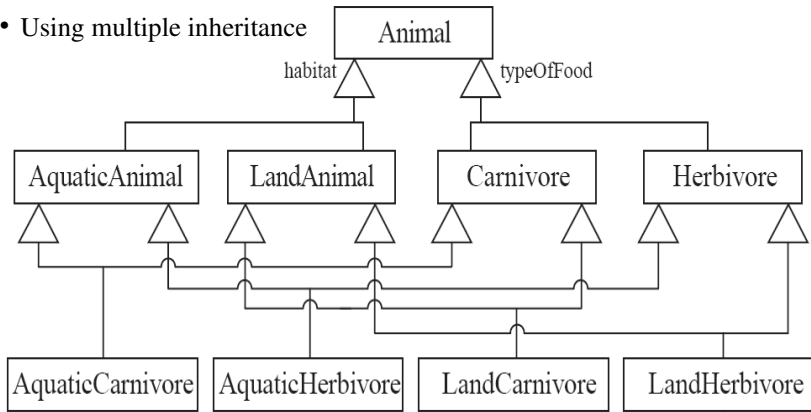
# Handling multiple discriminators

- Creating higher-level generalization



# Handling multiple discriminators

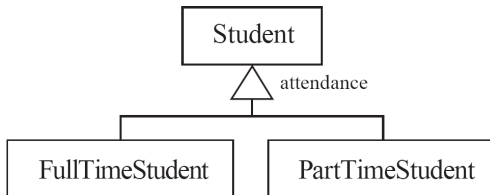
- Using multiple inheritance



- Using the Player-Role pattern (in Chapter 6)

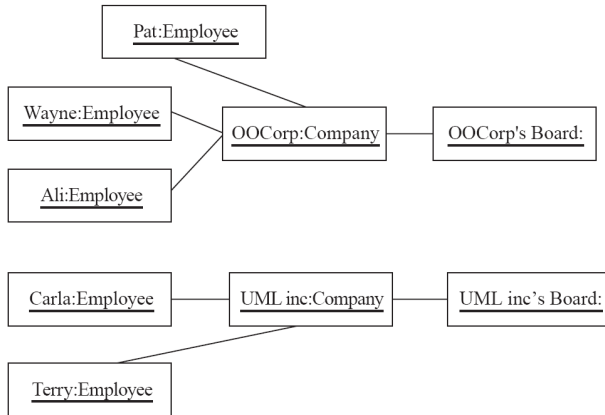
# Avoiding having instances change class

- An instance should never need to change class



## 5.5 Object Diagrams

- A *link* is an instance of an association
  - In the same way that we say an object is an instance of a class



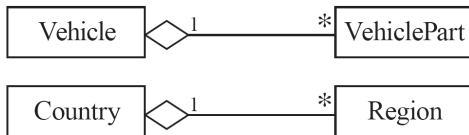
# Associations versus generalizations in object diagrams

- Associations describe the relationships that will exist between *instances* at run time.
  - When you show an instance diagram generated from a class diagram, there will be an instance of *both* classes joined by an association
- Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses



## 5.6 More Advanced Features: Aggregation

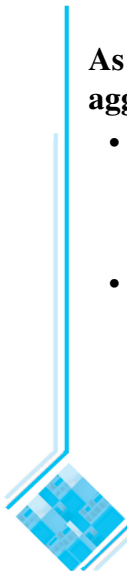
- Aggregations are special associations that represent ‘part-whole’ relationships.
  - The ‘whole’ side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named `isPartOf`



# When to use an aggregation

**As a general rule, you can mark an association as an aggregation if the following are true:**

- You can state that
  - the parts ‘are part of’ the aggregate
  - or the aggregate ‘is composed of’ the parts
- When something owns or controls the aggregate, then they also own or control the parts

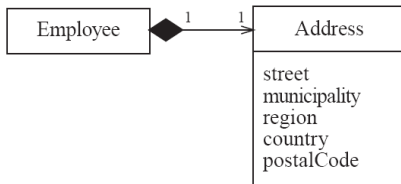
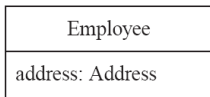


# Composition

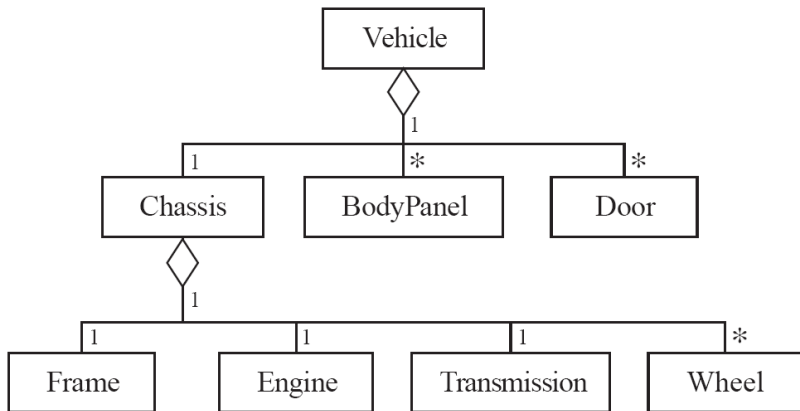
- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well



- Two alternatives for addresses



# Aggregation hierarchy



## Example (Something Like Tetris)

*We have a game field where a piece composed of squares falls down from the top of the field until it hits another piece or the bottom. Then the piece stops and stays there and the next piece comes from above. The pieces fall slowly, the player can rotate them by 90 degrees and move them left and right with the keyboard. If there is one row completely filled by pieces, this row is cut out of the playing area and all rows above are moved down by one row. The stones fall faster and faster the longer the game runs. The player loses if a new piece cannot enter the screen because it is filled with pieces.*

**Blackboard:** → **Associations, Aggregations,**  
**Composition in Something Like Tetris** ←

**Blackboard:** → **Multiplicity in Something Like Tetris** ←

# Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the entire process until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces
  - Apply design patterns (Chapter 6)

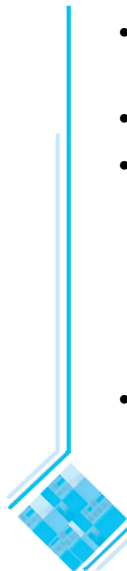
*Don't be too disorganized. Don't be too rigid either.*

# Identifying classes

- When developing a domain model you tend to *discover* classes
- When you work on the user interface or the system architecture, you tend to *invent* classes
  - Needed to solve a particular design problem
  - (Inventing may also occur when creating a domain model)
- Reuse should always be a concern
  - Frameworks
  - System extensions
  - Similar systems

# A simple technique for discovering domain classes

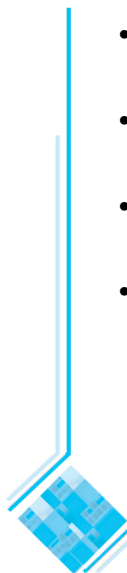
- Look at a source material such as a description of requirements
- Extract the *nouns* and *noun phrases*
- Eliminate nouns that:
  - are redundant
  - represent instances
  - are vague or highly general
  - not needed in the application
- Pay attention to classes in a domain model that represent *types of users* or other actors





# Identifying associations and attributes

- Start with classes you think are most **central** and important
- Decide on the clear and obvious data it must contain and its relationships to other classes.
- Work outwards towards the classes that are less important.
- Avoid adding many associations and attributes to a class
  - A system is simpler if it manipulates less information



# Tips about identifying and specifying valid associations

- An association should exist if a class
  - *possesses*
  - *controls*
  - *is connected to*
  - *is related to*
  - *is a part of*
  - *has as parts*
  - *is a member of*, or
  - *has as members*

some other class in your model

- Specify the multiplicity at both ends
- Label it clearly.

## Example

Marzipan (badem ezmesi) is a game, which is played by one player on a grid. In each grid cell there is one almond. The player can swap almonds of two neighboring cells, but only if then three almonds of the same type are in one row or column next to each other. After such a move all sequences of at least three almonds of the same type in a row or column are removed, and almonds in cells above empty cells are pulled down by gravity and move into these empty cells. Empty cells on top of the field are filled by random new almonds. If movement of almonds again causes three or more almonds of the same type to lie in one row or column, these almonds are also removed (and gravity fills fields). The score is the sum of squares of almonds removed in each move. (Making a move that indirectly removes many more almonds gives higher scores.) In the future we might want to add the following features: (A) the grid can have missing/blocked cells, (B) gravity changes after each move (up, left, right, down), (C) joker almonds combine with any other almond type, (D) almonds bombs remove adjacent almonds when they are removed.

**Question:** → Which Noun Phrases could/should be Classes?  
What are useful Associations and their Multiplicity? ←

## Java Programming Assignment

Marzipan (badem ezmesi) is a game, which is played by one player on a grid. In each grid cell there is one almond. The player can swap almonds of two neighboring cells, but only if then three almonds of the same type are in one row or column next to each other. After such a move all sequences of at least three almonds of the same type in a row or column are removed, and almonds in cells above empty cells are pulled down by gravity and move into these empty cells. Empty cells on top of the field are filled by random new almonds. If movement of almonds again causes three or more almonds of the same type to lie in one row or column, these almonds are also removed (and gravity fills fields). The score is the sum of squares of almonds removed in each move. (Making a move that indirectly removes many more almonds gives higher scores.) In the future we might want to add the following features: (A) the grid can have missing/blocked cells, (B) gravity changes after each move (up, left, right, down), (C) joker almonds combine with any other almond type, (D) almonds bombs remove adjacent almonds when they are removed.

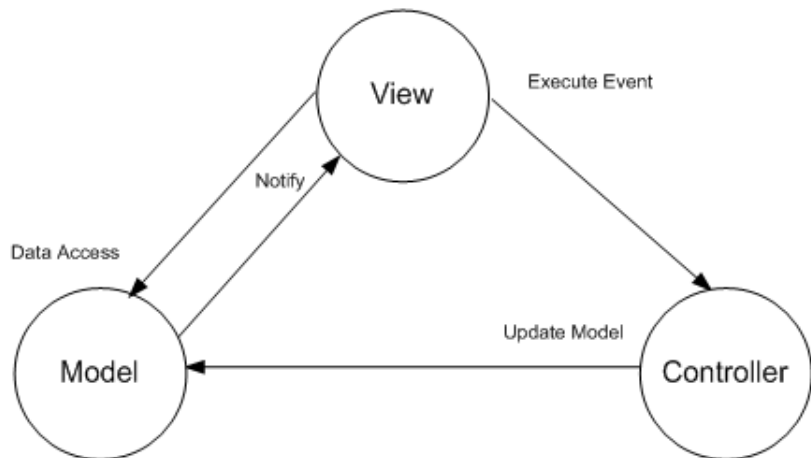
# Your Tasks

- ▶ Create a software design for this game:
  - ▶ Analyze the text.
  - ▶ Create one high-level and four detailed class/object/sequence diagrams for the Model of the software design. (Exactly 5 diagrams in total, you decide about type and level of detail.)
  - ▶ Use the MVC architecture in your design.
  - ▶ Explain/draw for at least one possible future enhancement (A,B,C,D) how it can be integrated into your design.
- ▶ Implement parts of the game in Java:
  - ▶ The field is shown in text mode, the game randomly plays legal moves (View = text, Controller = random moves, most programming is the Model).
  - ▶ Bonus: implement the view in graphics.
  - ▶ Bonus: implement a controller that allows user interaction.
  - ▶ Bonus: implement one or more enhancements.
- ▶ Hand in (A) diagrams, (B) explanation of enhancement, (C) source code, (D) JAR file, (E) usage instructions.

# Model View Controller Architecture

- ▶ Architecture for developing flexible applications of any kind
- ▶ Three core parts of the architecture:
  - ▶ **Model**: contains and controls the data of the application
  - ▶ **View**: presents the model to the user
  - ▶ **Controller**: allows interaction of the user with view and model
- ▶ Important concept: **separation of concerns**

## One possible view of Model-View-Controller



# Separation of Concerns

- ▶ Concerns can be separated on a high level ...
  - ▶ security: assuring permissions
  - ▶ user interface: interaction with user
  - ▶ storage: representing, storing, loading internal data
- ▶ Concerns can be separated on a low level ...
  - ▶ security: how to establish a secure channel
  - ▶ security: how to communicate over a secure channel
  - ▶ security: how to recover from failure of a secure channel

Another example:

- ▶ user interface: what is the behavior of a menu item
- ▶ user interface: what is the behavior of a symbol bar
- ▶ user interface: what is the behavior of a checkbox



# MVC - Example Word Processing - Model

Model of a word processor (MS Word, LibreOffice/OpenOffice Writer):

- ▶ document settings and how they can be adjusted (think about password-protected documents)
- ▶ sequence of sections of the document
- ▶ each section has sequence of paragraphs
- ▶ each paragraph has sequence of objects
- ▶ an object is a character or a picture or a form or a table
- ▶ ...

# MVC - Example Word Processing - View

View of a word processor:

- ▶ Print Preview View
  - ▶ everything looks as if it would be printed
- ▶ WYSIWYG (What You See Is What You Get) View
  - ▶ based on print preview
  - ▶ formatting marks/spelling correction visible
  - ▶
- ▶ Structur View
  - ▶ no page breaks
  - ▶ no page headers/footers
  - ▶ structure (bullets, numbers) more clearly visible
  - ▶ placeholders instead of pictures
- ▶ ...

# MVC - Example Word Processing - Controller

## Controller of a typical Word Processor

- ▶ Mouse interaction with text (mark, drag&drop, right click)
- ▶ Keyboard interaction with text (mark, cut, paste, write text, erase text)
- ▶ Menu bar + keyboard/mouse interaction with menu bar
- ▶ Keyboard shortcuts
- ▶ Scrolling the view

## MVC - Example Web Browser - Model

- ▶ stores DOM (Document Object Model) tree from the HTML
- ▶ stores CSS (Cascaded Style Sheets) formatting information
- ▶ stores Javascript code of the page
- ▶ stores Browser History and Bookmarks
- ▶ stores Cookies
- ▶ stores form information (e.g., Google search box)
- ▶ stores javascript internal state (e.g., for AJAX applications)
- ▶ can format the DOM using CSS
- ▶ can run javascript on the DOM

## MVC - Example Web Browser - View

- ▶ display formatted DOM to the user (render page)
- ▶ allow scrolling, zooming
- ▶ embed other applications (flash, video player)
- ▶ Possible alternative views:
  - ▶ Browsing View
  - ▶ Print View (output to printer)
  - ▶ Web Developer View (e.g., firebug in firefox)
    - allow highlighting of certain DOM elements
    - allow editing of DOM/CSS
  - ▶ Accessibility View (high contrast, large zoom, for visually disabled people)

# MVC - Example Web Browser - Controller

- ▶ Mouse scrolling, clicking (links, buttons)
- ▶ Keyboard Shortcuts
- ▶ Input via Gestures
- ▶ Viewing vs editing forms vs editing the document

# Another analogy for Model-View-Controller

**Model** = HTML

## css Zen Garden

### The Beauty of CSS Design

A demonstration of what can be accomplished  
this page.

Download the sample [html file](#) and [css file](#)

### The Road to Enlightenment

Littering a dark and dreary road lay the past

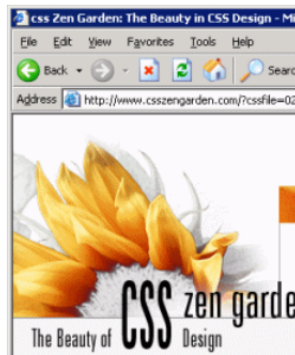
Today, we must clear the mind of past practice  
W3C, WaSP and the major browser creators

The css Zen Garden invites you to relax and re

**View** = CSS

```
body {  
    font: 12px/16px arial, helvetica, sans-serif;  
    color: #555;  
    background: url(bg_left.gif) repeat left top;  
    margin: 0;  
    padding: 0;  
}  
  
a {  
    text-decoration: none;  
    font-weight: bold;  
    color: #655;  
}  
  
a:hover {  
    text-decoration: none;  
    font-weight: bold;  
    color: #e60;  
}
```

**Controller** = Browser



## Example (Simple Dictionary)

*We want to build a dictionary application. Our dictionary is for two languages (English and Turkish) but we might want to extend it to more languages later. A word in one language can be translated to several words in the other language. Each translation can also have a comment (e.g., “technical”, or “law”). Translations can be marked as commonly used, archaic, informal language. Translations can have example sentences. The user can input a word and select which direction to search (searching in both directions is also possible). The search result shows first the commonly used, then unlabeled, then informal (argo) results. The user can create a profile with a login, in this profile the user can put words to a “training list”.*

**Blackboard:** → MVC for this example ←



## MVC - the best solution?

- ▶ User Selections are not in the model, but required in Controller and View ...
- ▶ User can scroll, which changes the View, but it is an event that should be managed by the Controller ...
- ▶ In general: input and output are often combined concerns  
⇒ View and Controller are often tightly coupled  
(Splitting them does not achieve much.)

# MVC - the best solution?

- ▶ User Selections are not in the model, but required in Controller and View ...
- ▶ User can scroll, which changes the View, but it is an event that should be managed by the Controller ...
- ▶ In general: input and output are often combined concerns  
⇒ View and Controller are often tightly coupled (Splitting them does not achieve much.)  
⇒ combine them? (Delegate-Model architecture)
- ▶ Many other Architectures for various purposes.
- ▶ Always a good idea:  
Separate change of data (simple consistency checks)  
from GUI (complex bells and whistles).

# MVC Questions

Question: → Can MVC be applied to BYS? ←

Question: → Can MVC be applied to a Chess program (complete information game). ←

Question: → Can MVC be applied to an online Card game program (incomplete information/chance game with server and client)? ←

Question: → Assume we have a web browser with tabs, and each tab has its own MVC architecture. How can we handle browser history and stored passwords? ←

## Java Programming Assignment

Marzipan (badem ezmesi) is a game, which is played by one player on a grid. In each grid cell there is one almond. The player can swap almonds of two neighboring cells, but only if then three almonds of the same type are in one row or column next to each other. After such a move all sequences of at least three almonds of the same type in a row or column are removed, and almonds in cells above empty cells are pulled down by gravity and move into these empty cells. Empty cells on top of the field are filled by random new almonds. If movement of almonds again causes three or more almonds of the same type to lie in one row or column, these almonds are also removed (and gravity fills fields). The score is the sum of squares of almonds removed in each move. (Making a move that indirectly removes many more almonds gives higher scores.) In the future we might want to add the following features: (A) the grid can have missing/blocked cells, (B) gravity changes after each move (up, left, right, down), (C) joker almonds combine with any other almond type, (D) almonds bombs remove adjacent almonds when they are removed.

# Your Tasks

- ▶ Create a software design for this game:
  - ▶ Analyze the text.
  - ▶ Create one high-level and four detailed class/object/sequence diagrams for the Model of the software design. (Exactly 5 diagrams in total, you decide about type and level of detail.)
  - ▶ Use the MVC architecture in your design.
  - ▶ Explain/draw for at least one possible future enhancement (A,B,C,D) how it can be integrated into your design.
- ▶ Implement parts of the game in Java:
  - ▶ The field is shown in text mode, the game randomly plays legal moves (View = text, Controller = random moves, most programming is the Model).
  - ▶ Bonus: implement the view in graphics.
  - ▶ Bonus: implement a controller that allows user interaction.
  - ▶ Bonus: implement one or more enhancements.
- ▶ Hand in (A) diagrams, (B) explanation of enhancement, (C) source code, (D) JAR file, (E) usage instructions.

# Interaction/Sequence Diagrams

- ▶ Interaction between Objects
- ▶ Behavior over time
- ▶ Collaboration

## 8.1 Interaction Diagrams

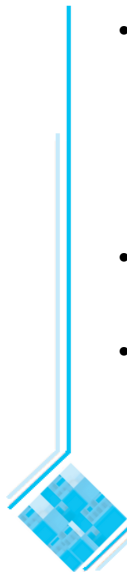
**Interaction diagrams are used to model the dynamic aspects of a software system**

- They help you to visualize how the system runs.
- An interaction diagram is often built from a use case and a class diagram.
  - The objective is to show how a set of objects accomplish the required interactions with an actor.



# Interactions and messages

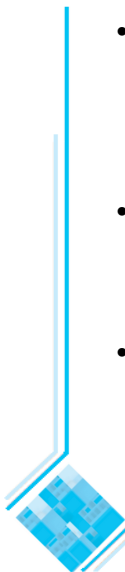
- Interaction diagrams show how a set of actors and objects communicate with each other to perform:
  - The steps of a use case, or
  - The steps of some other piece of functionality.
- The set of steps, taken together, is called an *interaction*.
- Interaction diagrams can show several different types of communication.
  - E.g. method calls, messages send over the network
  - These are all referred to as *messages*.





# Elements found in interaction diagrams

- Instances of classes
  - Shown as boxes with the class and object identifier underlined
- Actors
  - Use the stick-person symbol as in use case diagrams
- Messages
  - Shown as arrows from actor to object, or from object to object



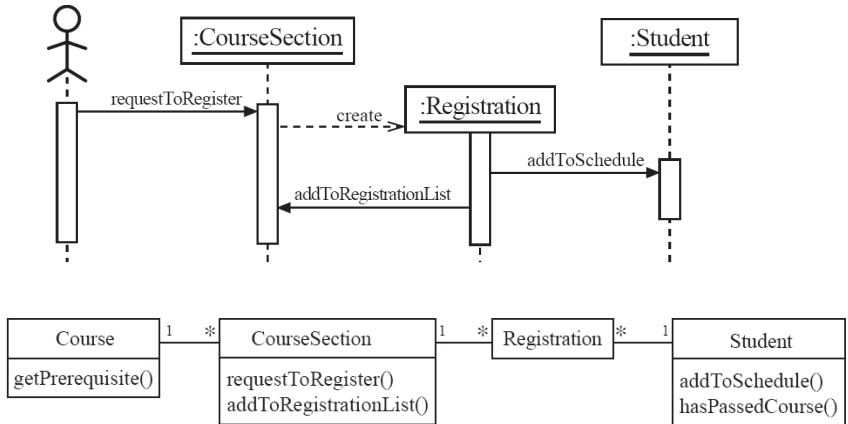
# Creating interaction diagrams

**You should develop a class diagram and a use case model before starting to create an interaction diagram.**

- There are two kinds of interaction diagrams:
  - *Sequence diagrams*
  - *Communication diagrams*



# Sequence diagrams – an example

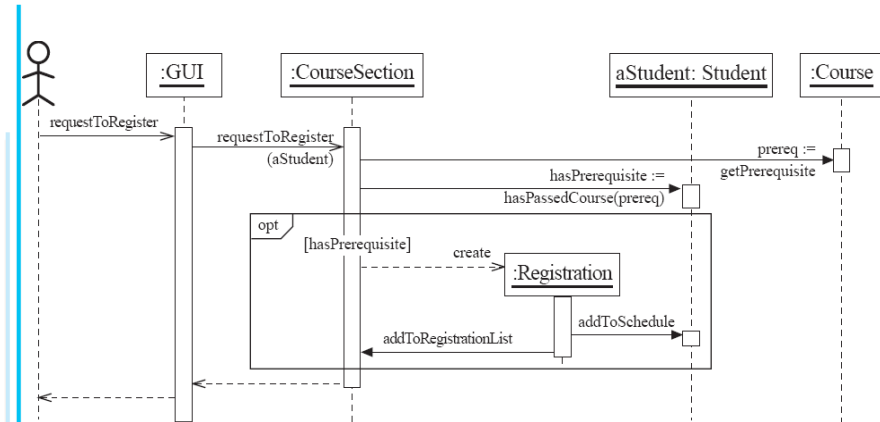


# Sequence diagrams

**A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task**

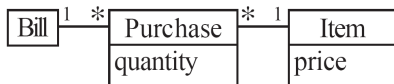
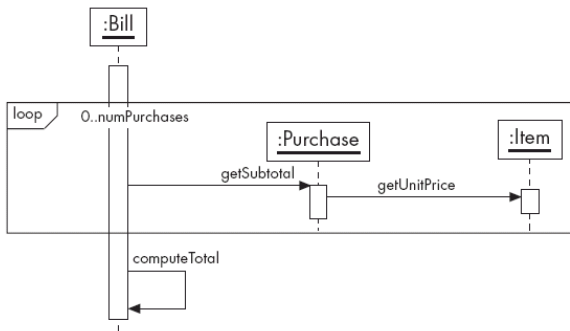
- The objects are arranged horizontally across the diagram.
- An actor that initiates the interaction is often shown on the left.
- The vertical dimension represents time.
- A vertical line, called a *lifeline*, is attached to each object or actor.
- The lifeline becomes a broad box, called an *activation box* during the *live activation* period.
- A message is represented as an arrow between activation boxes of the sender and receiver.
  - A message is labelled and can have an argument list and a return value.

# Sequence diagrams – same example, more details



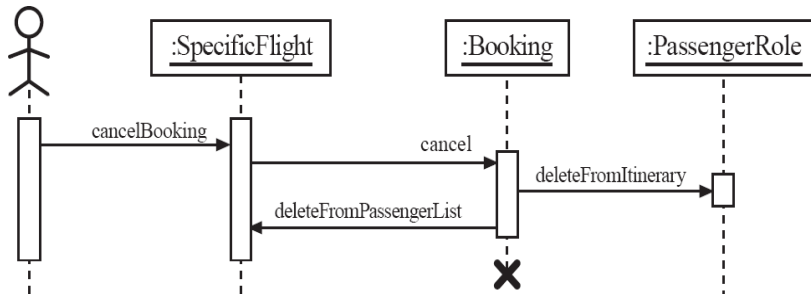
# Sequence diagrams – an example with replicated messages

- An *iteration* over objects is indicated by an asterisk preceding the message name



# Sequence diagrams – an example with object deletion

- If an object's life ends, this is shown with an X at the end of the lifeline



# Sequence Diagram Examples

**Blackboard:** → MVC Sequence Diagram for entering a character in Word Processor ←

**Blackboard:** → Online Chess Sequence Diagrams ←



## Midterm - Core Topics

- ▶ What are Challenges of Software Projects?
- ▶ What are Strategies for more successful Software Projects?
- ▶ What is UML and how can it be useful in Software Development?
- ▶ What are important properties of Object Oriented Languages?
- ▶ Explain Classes, Abstract Classes, Methods, Inheritance, Objects, Object Identity, Polymorphy, Association/Aggregation/Composition.
- ▶ Be able to interpret Class, Object, Sequence diagrams.
- ▶ Be able to create Class, Object, Sequence diagrams given a text description.
- ▶ Describe the Model View Controller architecture, its advantages and disadvantages.

## Midterm - Example Exercise

### Paper Shop

We want to have a software for managing orders and invoices in a paper shop. The shop sells printing and drawing paper. Each product has a price and a size (e.g., A4). Printing paper has a thickness, while drawing paper has a color. An order contains a date and a list of items with an amount. An invoice contains a date, a list of items with amount, an invoice number, and may contain a manual discount.

## Principle #1

*Minimize The Accessibility of Classes and Members*

## **The Meaning of Abstraction**

- Tony Hoare: “Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences.”
- Grady Booch: “An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.”
- Abstraction is one of the fundamental ways to deal with complexity
- An abstraction focuses on the outside view of an object and separates an object’s behavior from its implementation

## Encapsulation

- Grady Booch: “Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.”
- Craig Larman: “Encapsulation is a mechanism used to hide the data, internal structure, and implementation details of an object. All interaction with the object is through a public interface of operations.”
- Classes should be opaque
- Classes should not expose their internal implementation details

## Principle #2

*Favor Composition Over Inheritance*

## Composition

- Method of reuse in which new functionality is obtained by creating an object *composed of* other objects
- The new functionality is obtained by delegating functionality to one of the objects being composed
- Sometimes called *aggregation* or *containment*, although some authors give special meanings to these terms
- For example:
  - ⇒ Aggregation - when one object owns or is responsible for another object and both objects have identical lifetimes (GoF)
  - ⇒ Aggregation - when one object has a collection of objects that can exist on their own (UML)
  - ⇒ Containment - a special kind of composition in which the contained object is hidden from other objects and access to the contained object is only via the container object (Coad)

## Composition

- Composition can be:
  - ⇒ By reference
  - ⇒ By value
- C++ allows composition by value or by reference
- But in Java all we have are object references!



## Advantages/Disadvantages Of Composition

- Advantages:

- ⇒ Contained objects are accessed by the containing class solely through their interfaces
- ⇒ "Black-box" reuse, since internal details of contained objects are *not* visible
- ⇒ Good encapsulation
- ⇒ Fewer implementation dependencies
- ⇒ Each class is focused on just one task
- ⇒ The composition can be defined dynamically at run-time through objects acquiring references to other objects of the same type

- Disadvantages:

- ⇒ Resulting systems tend to have more objects
- ⇒ Interfaces must be carefully defined in order to use many different objects as composition blocks

## **Inheritance**

- Method of reuse in which new functionality is obtained by extending the implementation of an existing object
- The generalization class (the superclass) explicitly captures the common attributes and methods
- The specialization class (the subclass) extends the implementation with additional attributes and methods

## Advantages/Disadvantages Of Inheritance

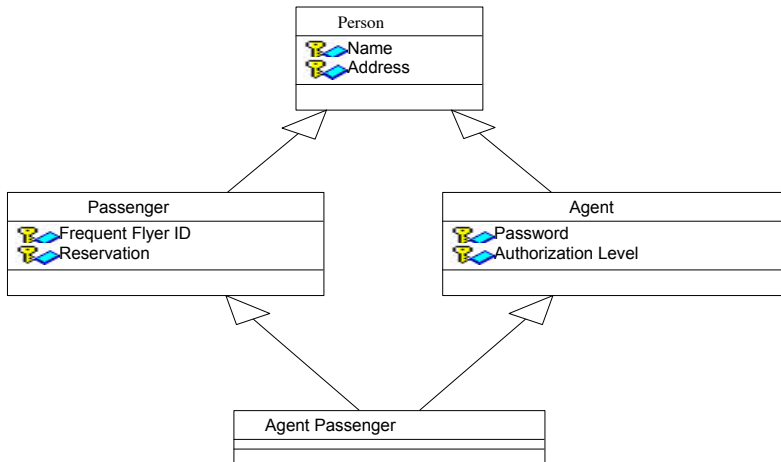
- Advantages:
  - ⇒ New implementation is easy, since most of it is inherited
  - ⇒ Easy to modify or extend the implementation being reused
- Disadvantages:
  - ⇒ Breaks encapsulation, since it exposes a subclass to implementation details of its superclass
  - ⇒ "White-box" reuse, since internal details of superclasses are often visible to subclasses
  - ⇒ Subclasses may have to be changed if the implementation of the superclass changes
  - ⇒ Implementations inherited from superclasses can not be changed at run-time

## Coad's Rules

Use inheritance only when all of the following criteria are satisfied:

- A subclass expresses "is a special kind of" and not "is a role played by a"
- An instance of a subclass never needs to become an object of another class
- A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass
- A subclass does not extend the capabilities of what is merely a utility class
- For a class in the actual Problem Domain, the subclass specializes a role, transaction or device

## Inheritance/Composition Example 1



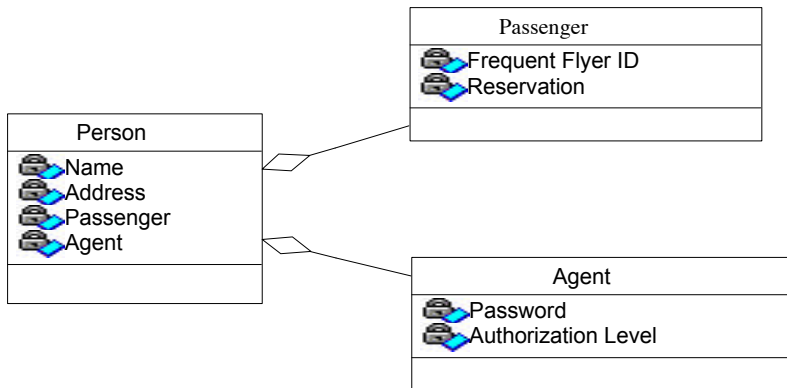
## Inheritance/Composition Example 1 (Continued)

- "Is a special kind of" not "is a role played by a"  
⇒ **Fail.** A passenger is a role a person plays. So is an agent.
- Never needs to transmute  
⇒ **Fail.** A instance of a subclass of Person could change from Passenger to Agent to Agent Passenger over time
- Extends rather than overrides or nullifies  
⇒ **Pass.**
- Does not extend a utility class  
⇒ **Pass.**
- Within the Problem Domain, specializes a role, transaction or device  
⇒ **Fail.** A Person is not a role, transaction or device.

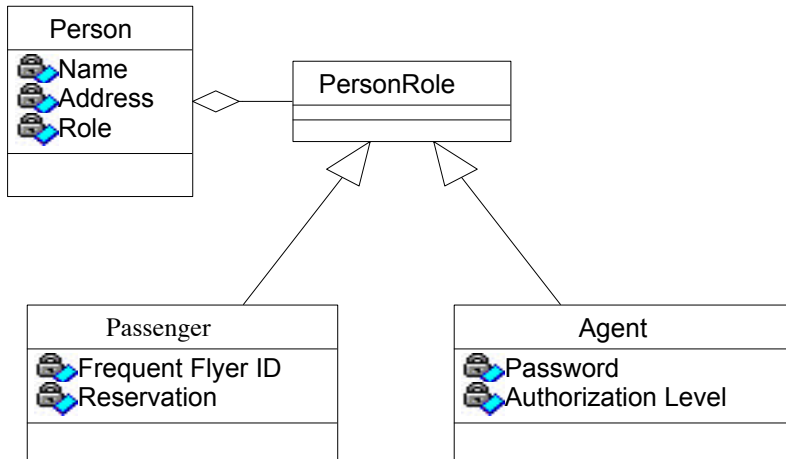
*Inheritance does not fit here!*

## Inheritance/Composition Example 1 (Continued)

*Composition to the rescue!*



## Inheritance/Composition Example 2



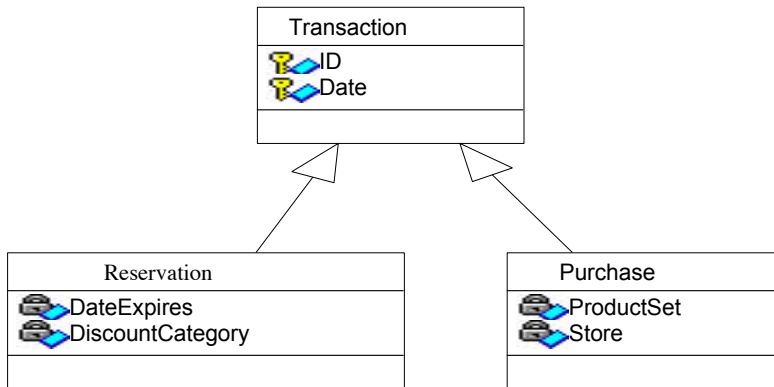


## Inheritance/Composition Example 2 (Continued)

- "Is a special kind of" not "is a role played by a"
  - ⇒ **Pass.** Passenger and agent are special kinds of person roles.
- Never needs to transmute
  - ⇒ **Pass.** A Passenger object stays a Passenger object; the same is true for an Agent object.
- Extends rather than overrides or nullifies
  - ⇒ **Pass.**
- Does not extend a utility class
  - ⇒ **Pass.**
- Within the Problem Domain, specializes a role, transaction or device
  - ⇒ **Pass.** A PersonRole is a type of role.

*Inheritance ok here!*

## Inheritance/Composition Example 3



## Inheritance/Composition Example 3 (Continued)

- "Is a special kind of" not "is a role played by a"  
⇒ **Pass.** Reservation and purchase are a special kind of transaction.
- Never needs to transmute  
⇒ **Pass.** A Reservation object stays a Reservation object; the same is true for a Purchase object.
- Extends rather than overrides or nullifies  
⇒ **Pass.**
- Does not extend a utility class  
⇒ **Pass.**
- Within the Problem Domain, specializes a role, transaction or device  
⇒ **Pass.** It's a transaction.

*Inheritance ok here!*

## Inheritance/Composition Summary

- Both composition and inheritance are important methods of reuse
- Inheritance was overused in the early days of OO development
- Over time we've learned that designs can be made more reusable and simpler by favoring composition
- Of course, the available set of composable classes can be enlarged using inheritance
- So composition and inheritance work together
- But our fundamental principle is:

*Favor Composition Over Inheritance*

## Principle #3

*Program To An Interface, Not An  
Implementation*

## Interfaces

- An *interface* is the set of methods one object knows it can invoke on another object
- An object can have many interfaces. (Essentially, an interface is a subset of all the methods that an object implements).
- A *type* is a specific interface of an object
- Different objects can have the same type and the same object can have many different types
- An object is known by other objects only through its interface
- In a sense, interfaces express "is a kind of" in a very limited way as "is a kind of that supports this interface"
- Interfaces are the key to pluggability!

# Scripting Languages - Why Scripting?

- ▶ 'If you have to do a task more than two times, write a program that does it for you.'
- ▶ How about renaming and moving files in a nontrivial way?
- ▶ How about a program that checks all professor's webpages and tells us if there are updates? Perhaps we also want to check our favorite webcomic?
- ▶ How about performing thousands of small experiments in an automatic way?

# Scripting Languages - Why Scripting?

- ▶ 'If you have to do a task more than two times, write a program that does it for you.'
  - ▶ How about renaming and moving files in a nontrivial way?
  - ▶ How about a program that checks all professor's webpages and tells us if there are updates? Perhaps we also want to check our favorite webcomic?
  - ▶ How about performing thousands of small experiments in an automatic way?
- ⇒ Use one or more 'scripts'!



# The Dark Side of Scripting

- ▶ Scripts are small programs, usually written fast, not built for eternity.
- ▶ Scripting languages usually do not require defining types and usually allow carelessness.
  - ▶ If the programmer was clever, it will work.
  - ▶ If not, it will just abort sometimes somewhere, perhaps with a new input that has not been anticipated, this can happen years after the code has been written.
  - ▶ A C++ or Java program would have generated an error during **compilation** in many cases.
- ▶ Scripts are less robust than industry-grade software.
- ▶ Scripting languages are good for lazy people.
- ▶ Very evil things are possible (e.g., change datatype at runtime).

## Python - Why Python?

- ▶ It is interpreted - it does not require compiling (run instantly).
- ▶ It does not require defining types of arguments ('duck-typing').
- ▶ It is fast enough for many professional business applications (and for sure fast enough for 'a quick hack').
- ▶ It is quite portable across MacOS, Windows, Linux.
- ▶ Python code is more readable than other scripting languages (bash, perl).
- ▶ There are many nice libraries (3D display, automata, complex mathematics, all kinds of databases, XML, regular expressions, subprocesses).
- ▶ If you start a project with Python and need a high-speed component, you can outsource functionality using a C API.

## Python - Why Python?

- ▶ It is interpreted - it does not require compiling (run instantly).
  - ▶ It does not require defining types of arguments ('duck-typing').
  - ▶ It is fast enough for many professional business applications (and for sure fast enough for 'a quick hack').
  - ▶ It is quite portable across MacOS, Windows, Linux.
  - ▶ Python code is more readable than other scripting languages (bash, perl).
  - ▶ There are many nice libraries (3D display, automata, complex mathematics, all kinds of databases, XML, regular expressions, subprocesses).
  - ▶ If you start a project with Python and need a high-speed component, you can outsource functionality using a C API.
- ⇒ Use Python! (But use the powers of Python wisely.)

## Python Example 1

```
def doit(f,t):  
    for n in range(f,t):  
        print(n)  
  
doit(1, 4)
```

## Python Example 1

```
def doit(f,t):  
    for n in range(f,t):  
        print(n)
```

```
doit(1, 4)
```

that's boring?

## Python Example 2

```
import sys

def quote(s):
    return "'"+s+"' "

print sys.argv
args = sys.argv[1:]
print args
args.sort()
mapped = map(quote, args)
print mapped
print '{'+"".join(mapped)+'}'
```

## Python Example 2

```
import sys

def quote(s):
    return "'"+s+"' "

print sys.argv
args = sys.argv[1:]
print args
args.sort()
mapped = map(quote, args)
print mapped
print '{'+"".join(mapped)+'}'
```

that's boring?

## Python Example 3

```
def doit(f,t):  
    for n in range(f,t):  
        print(n)  
  
import sys  
  
doit(1, 4)  
doit(sys.argv[1],argv[2])
```



## Python Example 3

```
def doit(f,t):  
    for n in range(f,t):  
        print(n)  
  
import sys  
  
doit(1, 4)  
doit(sys.argv[1], argv[2])
```

that's **wrong!** (who can find all four mistakes?)

## Python Unicode Example

```
# encoding=utf8
```

```
myname = "something turkish "  
print myname
```

that's good for **Turkish!** (and German)

## Python Unicode Example

```
# encoding=utf8
```

```
myname = "something turkish "  
print myname
```

that's good for **Turkish!** (and German)

If you forget the 'encoding=' line you get an ugly warning

Even if the only Turkish characters are in your **comments!**

## Type Coercion in Python

```
>>> 2 * 2
>>> 2.0 * 2
>>> 2 * 2.0
>>> 2.0 * 2.0
>>> 3 / 2
>>> 3.0 / 2
>>> 3 / 2.0
>>> 3.0 / 2.0
```

it automatically uses the **more complex** type

## Strings in Python

```
>>> "I'm a hero!"
>>> 'I\'m a hero!'
>>> """
... I'm a hero!
... """
>>> '''
... I'm a hero!
... '''
>>> print "C:\documents\trash" # \t => tabulator
>>> print r"C:\documents\trash"
```

There are ways to 'escape', 'multi-line' and 'raw' strings.

## String Operations

```
>>> print "foo"+"bar"  
>>> print "foobarbaz"[3], "foobarbaz"[3:5]  
>>> print "foobarbaz"[3:], "foobarbaz"[:3]  
>>> print "foobarbaz"[-1], "foobarbaz"[-3]  
>>> print "foobarbaz"[2:-3], len("foobarbaz")
```

Addressing from beginning and end!

```
>>> a = "foo"+"bar"  
>>> a[1] = "x" # ERROR!  
>>> b = bytearray(a)  
>>> b[1] = "x"  
>>> str(b)
```

Strings are **immutable**! (for efficiency)

## list, tuple, set, frozenset, dict

- ▶ list: sorted mutable list
- ▶ tuple: sorted immutable list
- ▶ set: mutable set
- ▶ frozenset: immutable set
- ▶ dict: key/value dictionary

Only **immutable** containers can be in sets and keys of dictionaries!

```
>>> l = [1, 2, 3] ; t = (1, 2, 3) ; t2 = (1,)
>>> s = set([1,2,3]) ; f = frozenset([1,2,3])
>>> d = {}
>>> d[t] = 'this is the tuple'
>>> d[f] = 'this is the frozenset'
>>> d[l] = 'this is the list' # ERROR
>>> d[s] = 'this is the set' # ERROR
```

## Functions and Arguments

```
def myFunction2(ar1 , ar2='default '):  
    print "we got", ar1 , 'and' , ar2
```

```
myFunction2(1)  
myFunction2(ar2='foo' , ar1=(1,3, 'foo' ,(1 ,)))
```

```
arguments = [17,18]  
myFunction2(*arguments)
```

```
arguments = (17,18)  
myFunction2(*arguments)
```

```
u = myFunction2  
u(17,18)  
print map(lambda x: u(x, 'inlambda' ), [1,2,3])
```



if / elif / else

```
import sys

if len(sys.argv) == 1:
    print "no arguments"
elif sys.argv[1] == 'hello':
    print "got hello"
else:
    print "got something else"

if len(sys.argv) != 1:
    print "got arguments!"
    if sys.argv[1] == 'hello':
        print "got hello"
    else: # perhaps this should be more left
        print "got nothing"
```

for / range / zip

```
for f in [1, 2, 3, 4, 5]:  
    print f  
for f in range(-5,5,2):  
    print f  
for f in zip( ['foo', 'bar', 'baz'], range(0,100) ):  
    print f  
for f in zip('foobazbaz', range(0,100)):  
    print f  
nums = { 1: 'bir', 2: 'iki', 3: 'uc', 4: 'dort' }  
for f in nums:  
    print f  
for f in nums.keys() + nums.values():  
    print f  
for f in nums.items():  
    print f
```

## Indentation matters!

```
for f in range(0,100):  
    if f < 40:  
        print "in if"  
        for g in range(f,f+3):  
            if g % 2 == 0:  
                print "we are at",f,g  
            else:  
                pass # TODO implement this case
```

## filter / lambda (functional programming)

```
grades = [ ('Jane', 97), ('John', 73),  
            ('Mary', 2), ('Alice', 77),  
            ('Peter', 0), ('Bob', 55), ]
```

```
def show(collection):  
    print "showing"  
    for g in collection:  
        print g
```

```
def failed(record):  
    return record[1] < 50
```

```
show(grades)  
show(filter(failed, grades))  
# anonymous function with argument x  
show(filter(lambda x: not failed(x), grades))  
show(filter(lambda y: y[1] > 50, grades))
```

## map / reduce (functional programming)

```
students = [ { 'name':      'John ', 'grade': 12 },
               { 'name':      'Jane ', 'grade': 24 },
               { 'name':      'Joe ', 'grade': 89 },
               { 'name': 'Jessica ', 'grade': 64 } ]
```

```
def show(collection):
    for g in collection:
        print g
```

```
def nice(stud):
    return "{0[name]} got {0[grade]} %".format(stud)
```

```
show(map(nice, students)) # nice list
# only names
```

```
show(map(lambda stud: stud['name'], students))
# sum of grades
print reduce(lambda x, st: x+st['grade'], students, 0)
```

and / or / not / in / True / False

```
>>> 1 == 2
>>> not 1 == 2
>>> not (1 == 2)
>>> 4 in range(0,10)
>>> 10 not in range(0,10)
>>> False and False or (1 == 2-1) # careful!
>>> False and (False or (1 == 2-1))
>>> not False and (False or (1 == 2-1))
```

## Comfortable Lists and Matrices (List Comprehension)

```
grades = [ ('Jane', 97), ('John', 73),  
            ('Mary', 2), ('Alice', 77),  
            ('Peter', 0), ('Bob', 55), ]
```

```
# list of names
```

```
print [ g[0] for g in grades ]
```

```
# pairs of numbers
```

```
print [ (x,2*x) for x in range(0,5) ]
```

```
# a fixed number 10 times
```

```
print [ 0 for x in range(0,10) ]
```

```
# zero 5x5 matrix
```

```
print [ [ 0 for y in range(0,5) ]  
        for x in range(0,5) ]
```

```
# triangle 4x5 matrix
```

```
print [ [ int(x < y) for y in range(0,5) ] \  
        for x in range(0,4) ]
```

## Garbage Collection / del

```
import os, psutil, random, gc
def showMem():
    # memory measurement
    process = psutil.Process(os.getpid())
    mem = process.get_memory_info()[0]/float(2**20)
    print "{mb:4.0f} MB and {obj} objects".format(
        mb=mem, obj=len(gc.get_objects()))

showMem()
onemil = [ [random.randint(0,100)]
            for x in range(1000*1000) ]
print "after creating onemil" ; showMem()
del(onemil) ; print "after del" ; showMem()
twomil = [ [random.randint(0,100)]
            for x in range(2*1000*1000) ]
print "after creating twomil" ; showMem()
twomil = 'foo!' ; print "assignment" ; showMem()
```



## Garbage Collection (Python, Java, ...)

- ▶ Objects are created explicitly by the program
- ▶ Objects can be destroyed explicitly by the user (del)
- ▶ Objects can be destroyed automatically (implicitly) when they are no longer used/reachable
  - 'Garbage Collection' / 'Garbage Collector'
- ▶ Danger: if objects point to each other in a circle
  - ▶ Every object is reachable from another object
  - ▶ Garbage Collection more difficult / impossible
  - ▶ Example: People and Contacts (I have you in my contacts, you have me in your contacts).
- ▶ Python might garbage-collect objects but keep the memory for the future (to reuse it).  
(Getting memory from the operating system takes time.)

## Copies and References

Why does my data change randomly?

```
students = [ { 'name': 'Juliet' },  
              { 'name': 'Jane' },  
              { 'name': 'Jessica' } ]
```

```
def show(collection):  
    print "showing"  
    for g in collection:  
        print g
```

*# create grading scheme*

```
scheme = { 'perc': None, 'letter': None }
```

```
for s in students: # s is a reference!
```

```
    s['grade'] = scheme
```

```
show(students) # try to use copy.deepcopy(scheme)!
```

*# set grade of one student*

```
students[0]['grade']['perc'] = 100
```

```
show(students)
```

## String Formatting (% and format)

Let's make it beautiful!

```
grades = [ ('Jane', 97), ('John', 73),  
            ('Mary', 2), ('Alice', 77),  
            ('Peter', 0), ('Bob', 55), ]  
  
def show(collection):  
    for g in collection:  
        print g  
  
def nice1(record):  
    return "{r[0]:<10} got {r[1]:>4} %".\   
        format(r=record)  
  
def nice2(record):  
    return "%10s got %4d %%" % record  
  
show(grades)  
show(map(nice1, grades))  
show(map(nice2, grades))
```

## Modules - finally we do Software Engineering!

- ▶ Let's say you have file `mymodule.py`

```
magicnumber = 123567
def superfunction(a,b,c,d,e):
    return 0
def main():
    print "this module does not do much"
if __name__ == '__main__':
    main()
```

- ▶ You can call your module as a program

```
$ python mymodule.py
```

- ▶ You can use your module in other code

```
>>> import mymodule
>>> dir(mymodule) # what can the module do?
>>> mymodule.magicnumber
>>> mymodule.superfunction(1,2,3,4,5)
>>> mymodule.main()
```

## More Import

```
>>> from mymodule import superfunction
>>> superfunction(1,2,3,4,5)
>>> mymodule.magicnumber # error
>>> import mymodule as mm
>>> mm.magicnumber # ok
>>> from mymodule import *
>>> magicnumber
>>> main()
```

## Objects and Classes

- ▶ Everything in Python behaves like an object
- ▶ Even classes are objects
- ▶ Don't worry about that!

```
>>> def foo(x=1):  
...     return x+1  
...  
>>> foo  
>>> foo()  
>>> lambda o: foo(o)  
>>> a = foo  
>>> a(7)  
>>> class Foo:  
...     pass  
...  
>>> Foo  
>>> Foo()
```

```

class Word:
    def __init__(self , form ):
        self.form = form
    def describe(self):
        print "Wordform", self.form
class Verb(Word):
    def __init__(self , form , arguments=[]):
        Word.__init__(self ,form)
        self.arguments = arguments
    def describe(self):
        Word.describe(self)
        print " with", self.arguments
words = []
words.append(Word('ama')) # create object
words.append(Verb('yazmak' , ['kim' , 'neyi']))
words.append(Verb('esmek'))
for w in words:
    w.describe()

```

## Instance and Class Variables

```
class Test:
    name = 'foo'
    def setName(self, name):
        self.name = name
t = Test()
print 't.name', t.name, 'Test.name', Test.name
print "=> setName"
t.setName('hello')
print 't.name', t.name, 'Test.name', Test.name
print "=> Test.name"
Test.name = 'bar'
print 't.name', t.name, 'Test.name', Test.name
print "=> t.name"
t.name = 'hi'
print 't.name', t.name, 'Test.name', Test.name
# del(t.name);del(Test.name);del(Test.setName) ?
```



## \_\_main\_\_ and other “builtin” magic

```
class Flexible:
    def __setattr__(self, name, value):
        print "setting", name, "to", value
    def __getattr__(self, name):
        print "sorry!, no attribute", name
        return self.whatever
    def whatever(self, *args):
        print "doing whatever with arguments ",args
def main():
    f = Flexible()
    f.hello(1)
    f.bar
    f.bar = "yes"
if __name__ == '__main__':
    main()
```

<https://docs.python.org/2/reference/datamodel.html#special-method-names>

## Exceptions: raise / try / except / finally

```
import sys
try:
    print int(sys.argv[1])/int(sys.argv[2])
except ZeroDivisionError as e:
    print "infinity is not a number!", e
except:
    print "oh yoo something unexpected!"
    import traceback
    print traceback.format_exc()
else:
    print "it seems there was no problem"
finally:
    print "good evening"

print "still alive"
raise Exception("the end is near")
print "still alive?"
```

## User-defined Exceptions

```
class VeryBadError(Exception):
    def __init__(self, info):
        Exception.__init__(self, info)
    def __str__(self):
        return "You have done bad things! ({})".\
            format(Exception.__str__(self))

def hm():
    raise VeryBadError("nothing here")

try:
    hm()
except Exception as e:
    # will catch VeryBadError (subclass of Exception)
    print "error?", e
hm() # will cause program to abort
```

## open and with

► Bad:

```
f = open('file.txt', 'w+')
f.write("hello file")
# f is still open
g = open('file.txt', 'r')
print repr(g.read())
```

► Good:

```
with open('file.txt', 'w+') as f:
    f.write("hello file")
# f is implicitly closed when leaving 'with' block
r = None
with open('file.txt', 'r') as g:
    r = g.read()
print repr(r)
```

## How to learn Python?

- ▶ Use my slides, repeat the experiments.  
⇒ read documentation/Google until you understand every line!
- ▶ Use the Python documentation (online)
- ▶ Use the interactive python shell (just run 'python').
- ▶ Use the Internet (Python is popular).
- ▶ Learn Python by solving small challenges:  
for example <http://projecteuler.net/>  
(I learned many great things from there.)  
((Can you find my profile?))

## Resources for learning Python

- ▶ We will use Python version 2!  
(Better support than 3, but not very different.)
- ▶ Python Tutorial:  
<https://docs.python.org/2/tutorial/index.html>  
Python Data Structures Tutorial:  
<https://docs.python.org/2/tutorial/datastructures.html>
- ▶ Python built-in types reference:  
<https://docs.python.org/2/library/stdtypes.html>
- ▶ Python built-in functions reference:  
<https://docs.python.org/2/library/functions.html>
- ▶ Python language reference:  
<https://docs.python.org/2/reference/index.html>
- ▶ Python standard library reference:  
<https://docs.python.org/2/library/index.html>

# Python Programming Assignment (Demo)

- ▶ Obtaining the Data
- ▶ Running the Assignment
- ▶ Measuring Time and Memory
- ▶ Python Interactive Shell
- ▶ Documentation (Tutorial, open, file, dict, tuple, for, \_\_next\_\_)

# Python Programming Assignment (1)

The program ‘analyze.py’ (see homepage) reads a file, splits it into words, counts which pair of words occurs how often, and which triple of words occurs how often, and then displays the most frequent pairs and triples of words. In this assignment, you will refactor this program according to the instructions below, and measure the effect of the changes. Please do the following **in exactly that order**. **Each step uses the result of the previous step**.

1. Find a project partner (this assignment must be done in a 2-person team).
2. Obtain ‘analyze.py’ and data (tiny English data, small German data, large Turkish data) from the course website. Run the program on English data, read the source code and understand how it works.
3. Refactor the program as follows:
  - Create a class NGramCounter(n) that can count n-grams (bigrams, trigrams, 4-grams, ...).
  - Replace BigramCounter and TrigramCounter by NGramCounter.
  - Check if the program works correctly and save it as ‘analyzeRefactor1NGram.py’.
4. Run ‘analyzeRefactor1NGram.py’ on all three data sets and observe the memory and time usage (for your report) (memory is printed by the function ‘showMemTime’).
5. Reduce the memory usage by deleting data when it is no longer needed (see the ‘del’ keyword). Observe time and memory usage after this change. Save that file as ‘analyzeRefactor2Delete.py’.
6. Refactor the program so that you never load the complete file. You can achieve this using the iterator interface:

```
f = open(filename, 'r')
for line in f:
    # do everything you need to do with 'line' here
    # (for example counting bigrams/trigrams)
    # (the line data will be deleted automatically by python)
```

Observe memory and time usage after this change. Save that file as ‘analyzeRefactor3Iterator.py’.

7. Refactor the program:



## Python Programming Assignment (2)

Observe memory and time usage after this change. Save that file as 'analyzeRefactor4More.py'.

7. Refactor the program:

- Create a function 'def displayKMostFrequentNGramsInFile(k,n,filename)' which opens the file, creates a n-gram counter class, counts the n-grams, and then prints the k most frequent n-grams (reuse the code in 'analyzeRefactor3Iterator').
- In your main function, do only the following: call that function three times: show the 30 most frequent 2-grams, show the 20 most frequent 3-grams, show the 15 most frequent 4-grams. (This will read the whole file three times, this is intentional.)
- Observe memory and time usage after this change. Save that file as 'analyzeRefactor4More.py'.

8. Put the 4 programs you created so far into a .zip archive.

9. Change your programs so that all of them compute and print only the top-5 3-grams and nothing else.

Run the programs and fill in the empty cells of the following table:

program	lines of code  (#)	resource usage					
		English data		German data		Turkish data	
		mem (max MB)	time (s)	mem (max MB)	time (s)	mem (max MB)	time (s)
analyze.py analyzeRefactor1NGram.py analyzeRefactor2Delete.py analyzeRefactor3Iterator.py analyzeRefactor4More.py							

10. Write a report (1 page) where you describe:

- The time and memory usage of steps 3, 4, 5, 6, and the table from step 8.
- Comment on the lines of code, memory, and time usage of the programs.
- Is there any duplicate code left which could be eliminated?

## Python Programming Assignment (3)

	(#)	(max MB)	(s)	(max MB)	(s)	(max MB)	(s)
analyze.py							
analyzeRefactor1NGram.py							
analyzeRefactor2Delete.py							
analyzeRefactor3Iterator.py							
analyzeRefactor4More.py							

10. Write a report (1 page) where you describe:
- The time and memory usage of steps 3, 4, 5, 6, and the table from step 8.
  - Comment on the lines of code, memory, and time usage of the programs.
  - Is there any duplicate code left which could be eliminated?
  - Which step above made the program more understandable/readable and which made it less understandable/readable?
  - What conclusions do you get from this assignment?
11. Send your report (PDF) and your .zip archive to [emel.kupcu@marmara.edu.tr](mailto:emel.kupcu@marmara.edu.tr) and [peter.schuller@marmara.edu.tr](mailto:peter.schuller@marmara.edu.tr) . Print your report and give it to Emel before the deadline (see course website).  
Include both of your names and both email addresses in the email!

## Python Programming Assignment (Technical Hints)

- ▶ 64 bit Windows: some libraries (numpy, numpy-mkl, scipy, matplotlib) work only if you install from an unofficial repository<sup>1</sup>

- ▶ Detailed memory profiling with memory\_profiler:

```
from memory_profiler import profile  
import psutil
```

```
@profile
```

```
def main():
```

```
    pass
```

(+) gives detailed memory statistics (-) program becomes slow  
Windows/python 2.7: if you get the error

```
error: Unable to find vcvarsall.bat
```

then install

```
psutil_2.1.2.win32_py2.7.exe1
```

<sup>1</sup> <http://www.lfd.uci.edu/~gohlke/pythonlibs/>

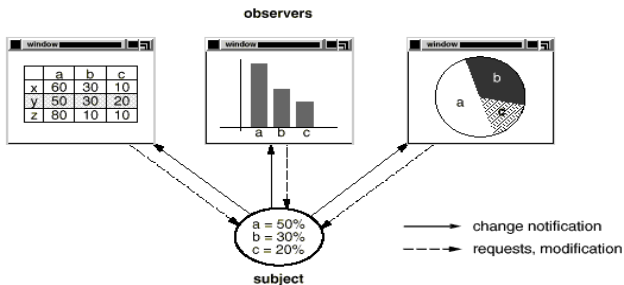
# The Observer Pattern

# The Observer Pattern

- Intent
  - ⇒ Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Also Known As
  - ⇒ Dependents, Publish-Subscribe, Model-View
- Motivation
  - ⇒ The need to maintain consistency between related objects without making classes tightly coupled

# The Observer Pattern

- Motivation



## The Observer Pattern

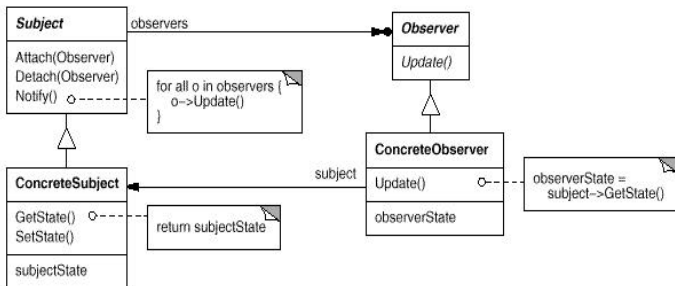
- Applicability

Use the Observer pattern in any of the following situations:

- ⇒ When an abstraction has two aspects, one dependent on the other.  
Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- ⇒ When a change to one object requires changing others
- ⇒ When an object should be able to notify other objects without making assumptions about those objects

# The Observer Pattern

- Structure





# The Observer Pattern

- Participants

- ⇒ Subject

- Keeps track of its observers
    - Provides an interface for attaching and detaching Observer objects

- ⇒ Observer

- Defines an interface for update notification

- ⇒ ConcreteSubject

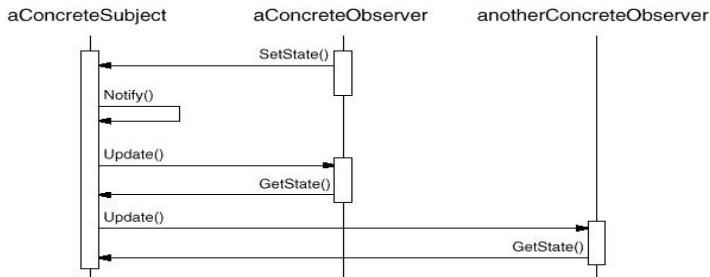
- The object being observed
    - Stores state of interest to ConcreteObserver objects
    - Sends a notification to its observers when its state changes

- ⇒ ConcreteObserver

- The observing object
    - Stores state that should stay consistent with the subject's
    - Implements the Observer update interface to keep its state consistent with the subject's

# The Observer Pattern

- Collaborations



# The Observer Pattern

- Consequences

⇒ Benefits

- Minimal coupling between the Subject and the Observer
  - Can reuse subjects without reusing their observers and vice versa
  - Observers can be added without modifying the subject
  - All subject knows is its list of observers
  - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
  - Subject and observer can belong to different abstraction layers
- Support for event broadcasting
  - Subject sends notification to all subscribed observers
  - Observers can be added/removed at any time

# The Observer Pattern

- Consequences
  - ⇒ Liabilities
    - Possible cascading of notifications
      - Observers are not necessarily aware of each other and must be careful about triggering updates
    - Simple update interface requires observers to deduce changed item

## The Observer Pattern

- Implementation Issues

- ⇒ How does the subject keep track of its observers?
  - Array, linked list
- ⇒ What if an observer wants to observe more than one subject?
  - Have the subject tell the observer who it is via the update interface
- ⇒ Who triggers the update?
  - The subject whenever its state changes
  - The observers after they cause one or more state changes
  - Some third party object(s)
- ⇒ Make sure the subject updates its state before sending out notifications
- ⇒ How much info about the change should the subject send to the observers?
  - Push Model - Lots
  - Pull Model - Very Little

# The Observer Pattern

- Implementation Issues

- ⇒ Can the observers subscribe to specific events of interest?
  - If so, it's publish-subscribe
- ⇒ Can an observer also be a subject?
  - Yes!
- ⇒ What if an observer wants to be notified only after several subjects have changed state?
  - Use an intermediary object which acts as a mediator
  - Subjects send notifications to the mediator object which performs any necessary processing before notifying the observers

# The Observer Pattern

- Sample Code
  - ⇒ We'll see some Java soon!
- Known Uses
  - ⇒ Smalltalk Model/View/Controller user interface framework
    - Model = Subject
    - View = Observer
    - Controller is whatever object changes the state of the subject
  - ⇒ Java 1.1 AWT/Swing Event Model
- Related Patterns
  - ⇒ Mediator
    - To encapsulate complex update semantics

## Java Implementation Of Observer

- We could implement the Observer pattern “from scratch” in Java
- But Java provides the Observable/Observer classes as built-in support for the Observer pattern
- The `java.util.Observable` class is the base Subject class. Any class that wants to be observed extends this class.
  - ⇒ Provides methods to add/delete observers
  - ⇒ Provides methods to notify all observers
  - ⇒ A subclass only needs to ensure that its observers are notified in the appropriate mutators
  - ⇒ Uses a Vector for storing the observer references
- The `java.util.Observer` interface is the Observer interface. It must be implemented by any observer class.



## The `java.util.Observable` Class

- `public Observable()`
  - ⇒ Construct an Observable with zero Observers
- `public synchronized void addObserver(Observer o)`
  - ⇒ Adds an observer to the set of observers of this object
- `public synchronized void deleteObserver(Observer o)`
  - ⇒ Deletes an observer from the set of observers of this object
- `protected synchronized void setChanged()`
  - ⇒ Indicates that this object has changed
- `protected synchronized void clearChanged()`
  - ⇒ Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change. This method is called automatically by `notifyObservers()`.

## The java.util.Observable Class

- public synchronized boolean hasChanged()
  - ⇒ Tests if this object has changed. Returns true if setChanged() has been called more recently than clearChanged() on this object; false otherwise.
- public void notifyObservers(Object arg)
  - ⇒ If this object has changed, as indicated by the hasChanged() method, then notify all of its observers and then call the clearChanged() method to indicate that this object has no longer changed. Each observer has its update() method called with two arguments: this observable object and the arg argument. The arg argument can be used to indicate which attribute of the observable object has changed.
- public void notifyObservers()
  - ⇒ Same as above, but the arg argument is set to null. That is, the observer is given no indication what attribute of the observable object has changed.

## The `java.util.Observer` Interface

- `public abstract void update(Observable o, Object arg)`
  - ⇒ This method is called whenever the observed object is changed. An application calls an observable object's `notifyObservers` method to have all the object's observers notified of the change.
  - ⇒ Parameters:
    - `o` - the observable object
    - `arg` - an argument passed to the `notifyObservers` method

## Observable/Observer Example

```
/**
 * A subject to observe!
 */
public class ConcreteSubject extends Observable {

    private String name;
    private float price;

    public ConcreteSubject(String name, float price) {
        this.name = name;
        this.price = price;
        System.out.println("ConcreteSubject created: " + name + " at "
            + price);
    }
}
```

## Observable/Observer Example (Continued)

```
public String getName() {return name;}

public float getPrice() {return price;}

public void setName(String name) {
    this.name = name;
    setChanged();
    notifyObservers(name);
}

public void setPrice(float price) {
    this.price = price;
    setChanged();
    notifyObservers(new Float(price));
}

}
```

## Observable/Observer Example (Continued)

```
// An observer of name changes.
public class NameObserver implements Observer {
    private String name;

    public NameObserver() {
        name = null;
        System.out.println("NameObserver created: Name is " + name);
    }
    public void update(Observable obj, Object arg) {
        if (arg instanceof String) {
            name = (String)arg;
            System.out.println("NameObserver: Name changed to " + name);
        } else {
            System.out.println("NameObserver: Some other change to
subject!");
        }
    }
}
```

## Observable/Observer Example (Continued)

```
// An observer of price changes.
public class PriceObserver implements Observer {
    private float price;

    public PriceObserver() {
        price = 0;
        System.out.println("PriceObserver created: Price is " + price);
    }
    public void update(Observable obj, Object arg) {
        if (arg instanceof Float) {
            price = ((Float)arg).floatValue();
            System.out.println("PriceObserver: Price changed to " +
                               price);
        } else {
            System.out.println("PriceObserver: Some other change to
                               subject!");
        }
    }
}
```

## Observable/Observer Example (Continued)

```
// Test program for ConcreteSubject, NameObserver and PriceObserver
public class TestObservers {
    public static void main(String args[]) {
        // Create the Subject and Observers.
        ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
        NameObserver nameObs = new NameObserver();
        PriceObserver priceObs = new PriceObserver();
        // Add those Observers!
        s.addObserver(nameObs);
        s.addObserver(priceObs);
        // Make changes to the Subject.
        s.setName("Frosted Flakes");
        s.setPrice(4.57f);
        s.setPrice(9.22f);
        s.setName("Sugar Crispies");
    }
}
```



## Observable/Observer Example (Continued)

- Test program output

```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0
PriceObserver: Some other change to subject!
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57
NameObserver: Some other change to subject!
PriceObserver: Price changed to 9.22
NameObserver: Some other change to subject!
PriceObserver: Some other change to subject!
NameObserver: Name changed to Sugar Crispies
```

## A Problem With Observable/Observer

- Problem:

⇒ Suppose the class which we want to be an observable is already part of an inheritance hierarchy:

```
class SpecialSubject extends ParentClass
```

⇒ Since Java does not support multiple inheritance, how can we have ConcreteSubject extend both Observable and ParentClass?

- Solution:

⇒ Use Delegation

⇒ We will have SpecialSubject contain an Observable object

⇒ We will delegate the observable behavior that SpecialSubject needs to this contained Observable object

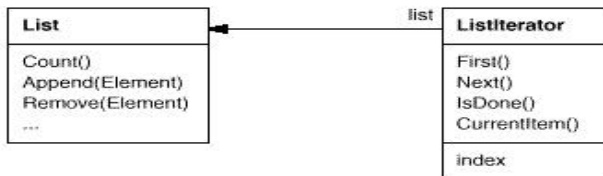
# The Iterator Pattern

# The Iterator Pattern

- Intent
  - ⇒ Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
  - ⇒ An *aggregate object* is an object that contains other objects for the purpose of grouping those objects as a unit. It is also called a *container* or a *collection*. Examples are a linked list and a hash table.
- Also Known As
  - ⇒ Cursor
- Motivation
  - ⇒ An aggregate object such as a list should allow a way to traverse its elements without exposing its internal structure
  - ⇒ It should allow different traversal methods
  - ⇒ It should allow multiple traversals to be in progress concurrently
  - ⇒ But, we really do not want to add all these methods to the interface for the aggregate

## Iterator Example 1

- List aggregate with iterator:



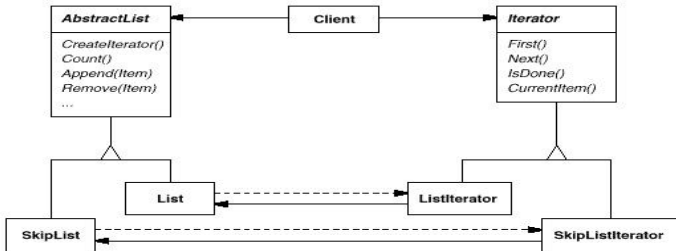
## Iterator Example 1 (Continued)

- Typical client code:

```
...  
List list = new List();  
...  
ListIterator iterator = new ListIterator(list);  
  
iterator.First();  
while (!iterator.IsDone()) {  
    Object item = iterator.CurrentItem();  
    // Code here to process item.  
    iterator.Next();  
}  
...
```

## Iterator Example 2

- Polymorphic Iterator



## Iterator Example 2 (Continued)

- Typical client code:

```
List list = new List();
SkipList skipList = new SkipList();
Iterator listIterator = list.CreateIterator();
Iterator skipListIterator = skipList.CreateIterator();
handleList(listIterator);
handleList(skipListIterator);
...
public void handleList(Iterator iterator) {
    iterator.First();
    while (!iterator.IsDone()) {
        Object item = iterator.CurrentItem();
        // Code here to process item.
        iterator.Next();
    }
}
```



## **The Iterator Pattern**

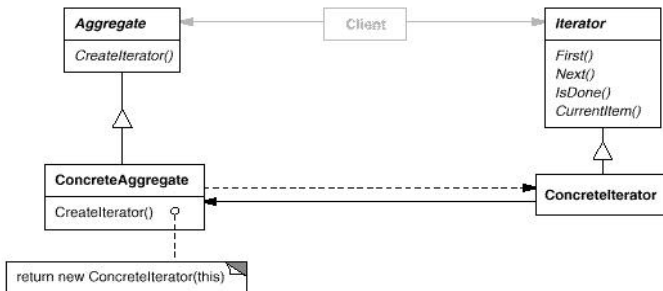
- **Applicability**

Use the Iterator pattern:

- ⇒ To support traversals of aggregate objects without exposing their internal representation
- ⇒ To support multiple, concurrent traversals of aggregate objects
- ⇒ To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration)

# The Iterator Pattern

- Structure



# The Iterator Pattern

- Participants

- ⇒ Iterator

- Defines an interface for accessing and traversing elements

- ⇒ ConcreteIterator

- Implements the Iterator interface
    - Keeps track of the current position in the traversal

- ⇒ Aggregate

- Defines an interface for creating an Iterator object (a factory method!)

- ⇒ ConcreteAggregate

- Implements the Iterator creation interface to return an instance of the proper ConcreteIterator

# The Iterator Pattern

- Consequences

- ⇒ Benefits

- Simplifies the interface of the Aggregate by not polluting it with traversal methods
    - Supports multiple, concurrent traversals
    - Supports variant traversal techniques

- ⇒ Liabilities

- None!

# The Iterator Pattern

- Implementation Issues

- ⇒ Who controls the iteration?

- The client => more flexible; called an external iterator

- The iterator itself => called an internal iterator

- ⇒ Who defines the traversal algorithm?

- The iterator => more common; easier to have variant traversal techniques

- The aggregate => iterator only keeps state of the iteration

- ⇒ Can the aggregate be modified while a traversal is ongoing? An iterator that allows insertion and deletions without affecting the traversal and without making a copy of the aggregate is called a *robust iterator*.

- ⇒ Should we enhance the Iterator interface with additional operations, such as previous()?

# Iterator in Python

- ▶ One method needs to be defined for container objects to provide iteration support:

```
container.__iter__()
```

Return an iterator object.

- ▶ The iterator objects are required to support the following two methods:

```
iterator.__iter__()
```

Return the iterator object itself. This is required to allow both containers and iterators to be used with the `for` and `in` statements.

```
iterator.next()
```

Return the next item from the container. If there are no further items, raise the `StopIteration` exception.

## Implicit Iterators in Python

```
l = [ 1, 2, 3, 4 ]  
l = list(1, 2, 3, 4)  
for item in l:  
    print item
```

```
d = set([1, 2, 3, 3, 4])  
for item in d:  
    print item
```

```
f = open("file", "r")  
for line in f:  
    print line
```

## Custom Iterators in Python

```
class Foolterator:  
    def __iter__(self):  
        return self  
    def next():  
        return "foo"
```

```
it = Foolterator()  
for item in it:  
    print item
```



## The Enumeration Interface

- `public abstract boolean hasMoreElements()`
  - ⇒ Returns true if this enumeration contains more elements; false otherwise
- `public abstract Object nextElement()`
  - ⇒ Returns the next element of this enumeration
  - ⇒ Throws: `NoSuchElementException` if no more elements exist
- Correspondences:
  - ⇒ `hasMoreElements()`  $\Rightarrow$  `IsDone()`
  - ⇒ `nextElement()`  $\Rightarrow$  `Next()` followed by `CurrentItem()`
  - ⇒ Note that there is no `First()`. `First()` is done automatically when the Enumeration is created.

## Enumeration Example

- Test program:

```
import java.util.*;

public class TestEnumeration {

    public static void main(String args[]) {
        // Create a Vector and add some items to it.
        Vector v = new Vector();
        v.addElement(new Integer(5));
        v.addElement(new Integer(9));
        v.addElement(new String("Hi, There!"));

        // Traverse the vector using an Enumeration.
        Enumeration ev = v.elements();
        System.out.println("\nVector values are:");
        traverse(ev);
    }
}
```

## Enumeration Example (Continued)

```
// Now create a hash table and add some items to it.
Hashtable h = new Hashtable();
h.put("Bob", new Double(6.0));
h.put("Joe", new Double(18.5));
h.put("Fred", new Double(32.0));

// Traverse the hash table keys using an Enumeration.
Enumeration ekeys = h.keys();
System.out.println("\nHash keys are:");
traverse(ekeys);

// Traverse the hash table values using an Enumeration.
Enumeration evalues = h.elements();
System.out.println("\nHash values are:");
traverse(evalues);
}
```

## Enumeration Example (Continued)

```
private static void traverse(Enumeration e) {  
    while (e.hasMoreElements()) {  
        System.out.println(e.nextElement());  
    }  
}  
  
}
```

## Enumeration Example (Continued)

- Test program output:

**Vector values are:**

**5**

**9**

**Hi, There!**

**Hash keys are:**

**Joe**

**Fred**

**Bob**

**Hash values are:**

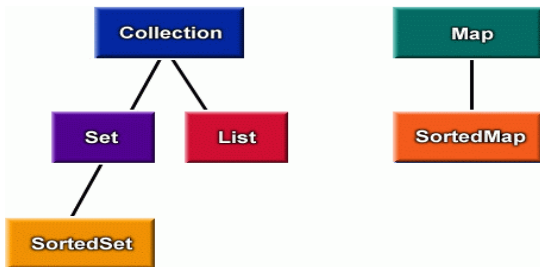
**18.5**

**32.0**

**6.0**

## The Java Collections Framework Interfaces

- Interfaces in the Java Collections Framework:



## The Collection Interface

- The java.util.Collection interface:

```
public interface Collection {  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(Object element);  
    boolean remove(Object element);  
    Iterator iterator();  
    ...  
}
```

## The Iterator Interface

- The Iterator object is similar to an Enumeration object with two differences:
  - ⇒ Method names have changed:
    - hasNext() instead of hasMoreElements()
    - next() instead of nextElement()
  - ⇒ Iterator is more robust than Enumeration in that it allows (with certain constraints) the removal (and in some cases addition) of elements from the underlying collection during the iteration
- The java.util.Iterator interface:

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```



## The List Interface

- java.util.List interface:

```
public interface List extends Collection {  
    Object get(int index);  
    Object set(int index, Object element);  
    void add(int index, Object element);  
    Object remove(int index);  
    boolean addAll(int index, Collection c);  
    int indexOf(Object o);  
    int lastIndexOf(Object);  
    ListIterator listIterator();  
    ListIterator listIterator(int index);  
    List subList(int from, int to);  
}
```

## The ListIterator Interface

- java.util.ListIterator interface:

```
public interface ListIterator extends Iterator {  
    boolean hasNext();  
    Object next();  
    boolean hasPrevious();  
    Object previous();  
    int nextIndex();  
    int previousIndex();  
    void remove();  
    void set(Object o);  
    void add(Object o);  
}
```

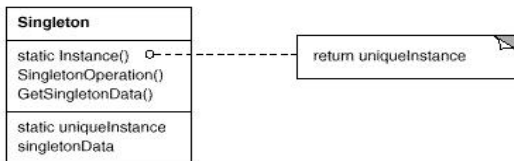
# The Singleton Pattern

# The Singleton Pattern

- Intent
  - ⇒ Ensure a class only has one instance, and provide a global point of access to it
- Motivation
  - ⇒ Sometimes we want just a single instance of a class to exist in the system
  - ⇒ For example, we want just one window manager. Or just one factory for a family of products.
  - ⇒ We need to have that one instance easily accessible
  - ⇒ And we want to ensure that additional instances of the class can not be created

# The Singleton Pattern

- Structure



- Consequences

⇒ Benefits

- Controlled access to sole instance
- Permits a variable number of instances

## Singleton Implementation

- OK, so how do we implement the Singleton pattern?
- We'll use a static method to allow clients to get a reference to the single instance and we'll use a private constructor!

```
/**
 * Class Singleton is an implementation of a class that
 * only allows one instantiation.
 */
public class Singleton {

    // The private reference to the one and only instance.
    private static Singleton uniqueInstance = null;

    // An instance attribute.
    private int data = 0;
```

## Singleton Implementation

```
/**
 * Returns a reference to the single instance.
 * Creates the instance if it does not yet exist.
 * (This is called lazy instantiation.)
 */
public static Singleton instance() {
    if(uniqueInstance == null) uniqueInstance = new Singleton();
    return uniqueInstance;
}

/**
 * The Singleton Constructor.
 * Note that it is private!
 * No client can instantiate a Singleton object!
 */
private Singleton() {}

// Accessors and mutators here!
}
```

## Singleton Implementation

- Here's a test program:

```
public class TestSingleton {  
  
    public static void main(String args[]) {  
        // Get a reference to the single instance of Singleton.  
        Singleton s = Singleton.instance();  
  
        // Set the data value.  
        s.setData(34);  
        System.out.println("First reference: " + s);  
        System.out.println("Singleton data value is: " +  
                             s.getData());  
    }  
}
```



## Singleton Implementation

```
// Get another reference to the Singleton.  
// Is it the same object?  
s = null;  
s = Singleton.instance();  
System.out.println("\nSecond reference: " + s);  
System.out.println("Singleton data value is: " +  
                    s.getData());  
}  
}
```

- And the test program output:

```
First reference: Singleton@1cc810  
Singleton data value is: 34
```

```
Second reference: Singleton@1cc810  
Singleton data value is: 34
```

## Singleton Implementation

- Note that the singleton instance is only created when needed. This is called *lazy instantiation*.
- Thought experiment: What if two threads concurrently invoke the instance() method? Any problems?
- Yup! Two instances of the singleton class could be created!
- How could we prevent this? Several methods:
  - ⇒ Make the instance() synchronized. Synchronization is expensive, however, and is really only needed the first time the unique instance is created.
  - ⇒ Do an eager instantiation of the instance rather than a lazy instantiation.

## Singleton Implementation

- Here is Singleton with eager instantiation.
- We'll create the singleton instance in a static initializer. This is guaranteed to be thread safe.

```
/**
 * Class Singleton is an implementation of a class that
 * only allows one instantiation.
 */
public class Singleton {

    // The private reference to the one and only instance.
    // Let's eagerly instantiate it here.
    private static Singleton uniqueInstance = new Singleton();

    // An instance attribute.
    private int data = 0;
```

## Singleton Implementation

```
/**
 * Returns a reference to the single instance.
 */
public static Singleton instance() {
    return uniqueInstance;
}

/**
 * The Singleton Constructor.
 * Note that it is private!
 * No client can instantiate a Singleton object!
 */
private Singleton() {}

// Accessors and mutators here!
}
```

## Singleton in Python

- ▶ Modules are singletons, nothing else is a singleton
- ▶ Assume a module 'tinymodule.py'

```
print "loading module"  
v = 10
```

- ▶ Then in the shell

```
>>> import tinymodule  
loading tinymodule  
>>> tinymodule.v  
10  
>>> tinymodule.v = 15  
>>> import tinymodule  
>>> tinymodule.v  
15
```

# The Visitor Pattern

# The Visitor Pattern

- Intent

⇒ Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

- Motivation

⇒ Consider a compiler that parses a program and represents the parsed program as an abstract syntax tree (AST). The AST has many different kinds of nodes, such as Assignment , Variable Reference, and Arithmetic Expression nodes.

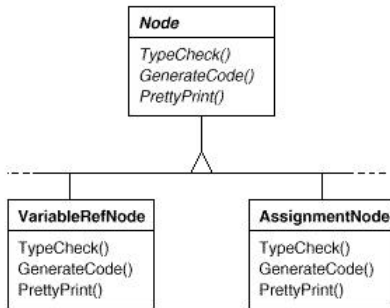
⇒ Operations that one would like to perform on the AST include:

- Checking that all variables are defined
- Checking for variables being assigned before they are used
- Type checking
- Code generation
- Pretty printing/formatting

# The Visitor Pattern

- Motivation

- ⇒ These operations may need to treat each type of node differently
- ⇒ One way to do this is to define each operation in the specific node class





# The Visitor Pattern

- Motivation

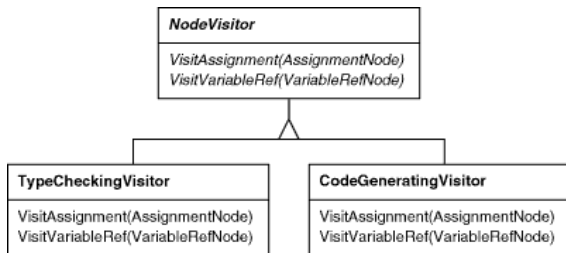
⇒ Problems with this approach:

- Adding new operations requires changes to all of the node classes
- It can be confusing to have such a diverse set of operations in each node class.  
For example, mixing type-checking code with pretty-printing code can be hard to understand and maintain.

⇒ Another solution is to encapsulate a desired operation in a separate object, called a visitor. The visitor object then traverses the elements of the tree. When an tree node "accepts" the visitor, it invokes a method on the visitor that includes the node type as an argument. The visitor will then execute the operation for that node - the operation that used to be in the node class.

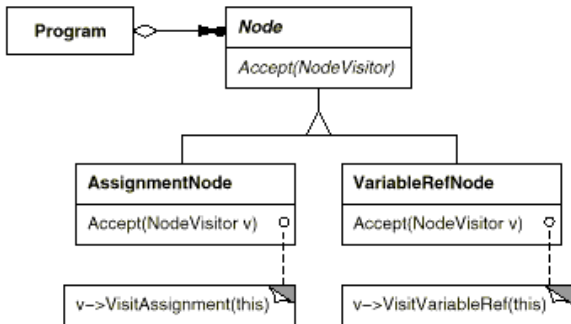
# The Visitor Pattern

- Motivation



# The Visitor Pattern

- Motivation



# The Visitor Pattern

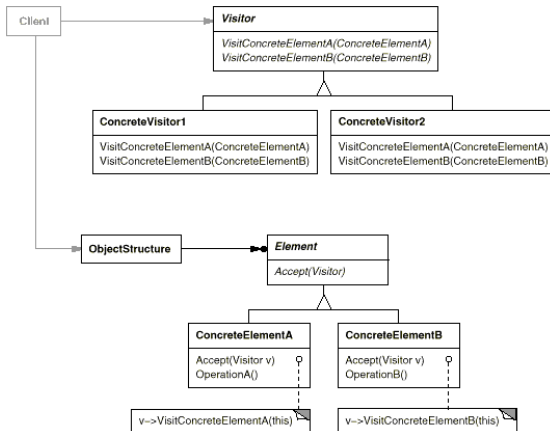
- Applicability

Use the Visitor pattern in any of the following situations:

- ⇒ When many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations
- ⇒ When the classes defining the object structure rarely change, but you often want to define new operations over the structure. (If the object structure classes change often, then it's probably better to define the operations in those classes.)
- ⇒ When an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes

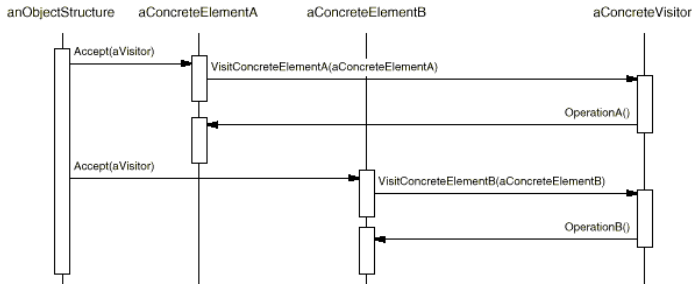
# The Visitor Pattern

- Structure



# The Visitor Pattern

- Collaborations



# The Visitor Pattern

- Consequences

- ⇒ Benefits

- Adding new operations is easy
    - Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor. Unrelated sets of behavior are partitioned in their own visitor subclasses.
    - Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would have to be passed as extra arguments to the operations that perform the traversal.

- ⇒ Liabilities

- Adding new ConcreteElement classes is hard. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class.
    - The ConcreteElement interface must be powerful enough to let visitors do their job. You may be forced to provide public operations that access an element's internal state, which may compromise its encapsulation.