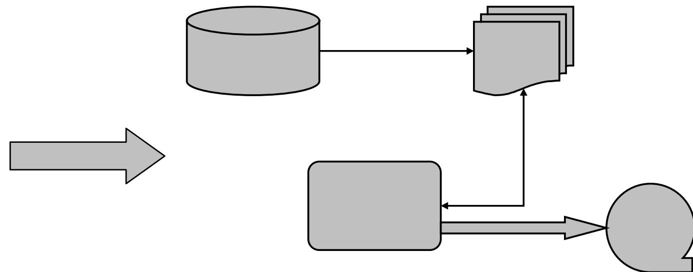
# Reverse Engineering and Code Annotations

```
**
 * @has 1..* Member * Student
 * @composed 1..* Has 1..* Department
 */
class School {
        Name name;
        String address;
        Number phone;
        …
}
```




Filippo Ricca
ITC-Irst
Povo (Trento), Italy
ricca@itc.it




# ITC-Irst

**ITC-irst** is a public research center of the Autonomous Province of Trento, Italy. It was founded in 1976. For nearly three decades, the Center has been conducting research in the areas of Information Technologies, Microsystems, and Physical Chemistry of Surfaces and Interfaces.

# STAR Group

- **Background**
  - Static code analysis and restructuring.
  - Reverse Engineering (CANTO)
  - Software Testing

- **Current activities**
  - Refactoring
  - Aspect Oriented Code
  - Analysis, Testing and Restructuring of Web applications
  - Code Annotations



# Reverse Engineering

- **Reverse engineering** is the process of taking something (a device, an electrical component, a car, a software, …) apart and analyzing its working in details, usually with the intention to construct a new device or program that does the same thing.
- Reverse engineering is used often **by military**, in order to copy other nations' technology.

Examples of military reverse-engineered projects include:

- Soviet Union reverse-engineered Tu-4 Bull bomber from United States B-29
- Soviet Union personal computer AGATHA was reverse-engineered from the Apple II
- North Korea reverse-engineered the Russian missile Scud Bs to make their own Scud Mod A

# Legacy systems

- They were implemented years ago.
- Their technology became obsolete (languages, coding style, hardware, …).
- They have been maintained for a long time.
- Their documentation (if it exists) became obsolete.
- Original authors are not available.
- Maintenance is difficult .
- They contain business rules not recorded elsewhere.
- They cannot be easly replaced.
- They represent a large investiment.

…



# Legacy dilemma

What should we do with legacy code?

- to build the new system from scratch
- trying to understand the legacy code and to try to reconstitute it in a new form

# Forward and Reverse Engineering

- **Forward engineering** is the traditional process of moving from high-level abstractions to the physical implementation of a system.

Requirements → Design → Implementation

- **Reverse engineering** is a process that helps understanding the system. It is a **process of examination not a process of change or replication**.

Requirements ← Design ← Implementation

Astract Code Representation ← Code



# Failure or success

Is Reverse Engineering of legacy systems doomed to failure?

- Answering this question requires an exact definition of reverse engineering.
- The answer is higly dependent on the specific goal of the reverse engineering.

# Definition (Strong)

The process of deriving formal specifications from the source code of a legacy system, where these specifications can be used to forward engineering a new implementation of that System.

Requirements ⇐ Design ⇐ Implementation

There are several assumptions underlying this definition:

1. The process is completely automatic
2. These specifications are at a sufficiently abstract level so that the system can be implemented in a new language or recoded in a more maintainable way.
3. The time and effort required to derive the specifications is less that starting from scratch.



# State of the art

- The current state of art in real-world reverse engineering is far from this goal. There are a lot of tools that help the engineering **to understand better the software** but not exist tools that, completely automatically, derive abstract formal specifications from code.

Specifications ⇐ Code NO!

# Tools available

1. Pretty printers (ex. SeeSoft)
2. Diagram generator (software views: flowcharts, data flow diagrams, call graph, …)
3. Embedded comments extractor (ex. Javadoc)
4. Software metrics
5. Design recovery tools (ex. UML class diagram extractor)
6. Others …



# Definition (Weak)

The automated or assisted process of deriving a **knowledge base** describing a legacy code from its source code.

Knowledge base ⟸ Code

1. The goal of reverse engineering is extracting a knowledge base and not a complete set of specifications
2. The process is not completely automated (human assistance)
3. Reverse engineering is successful if the cost of extracting information about legacy systems plus effort to arrive specifications is less than the cost of starting from scratch.

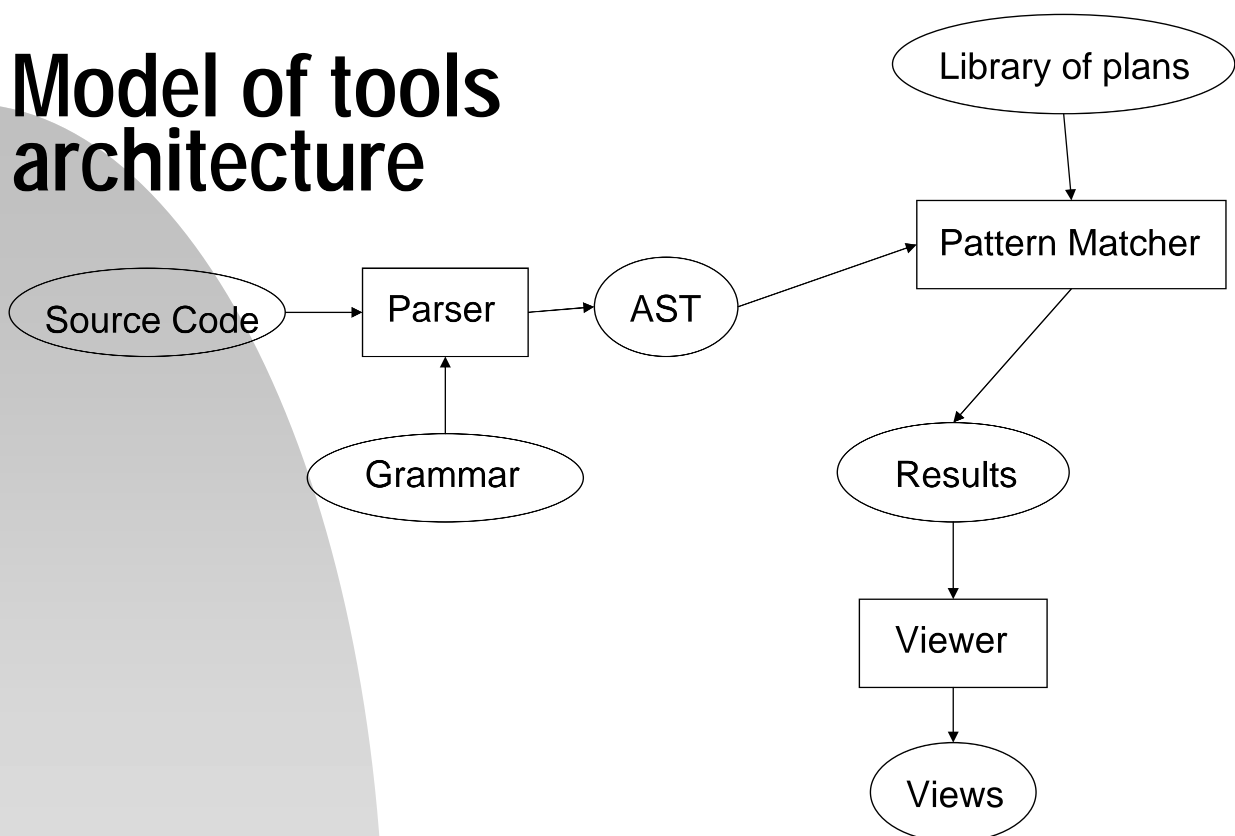
# What can be automatically extracted?

- Whether is possible to automatically extract specifications from code is an open question.
- Numerous current researches aim at extracting design information at various levels of abstractions.

- These approachs generally match code patterns (plans) against the code to recognize high-level abstractions (programming concepts, architectural concepts, domain concepts).



# Model of tools architecture



## Agile Processes and Reverse Engineering

- **No design phase** is prescribed, except for *quick design sessions*, the output of which serves just communication and discussion purposes, but is by no means a persistent artifact that documents the system.
- Different from UML-based processes.
- Strong statement: "***the source code is the design***".
- **Reverse engineering** can help in the *Program Comprehension* phase that is useful for:
  - Refactoring
  - Maintenance
  - ….



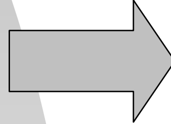
## Reverse Engineering Tools

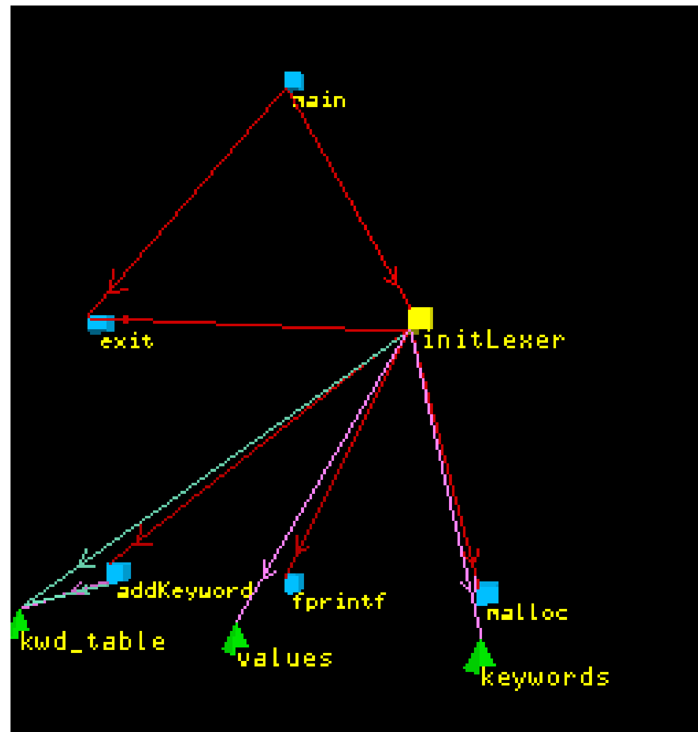
- We need to be realistic about what outcomes to expect.
- There seems to be general agreement that, in practice, reverse engineering of legacy code **is at least quite laborious and difficult.**
- The old notion of reverse enginnering may well be doomed to failure.
- **Cooperative extraction** (automated extraction tools + programmers) may be a good solution.

# Design Recovery Tools (1)

Imagix *tool*

'C code'



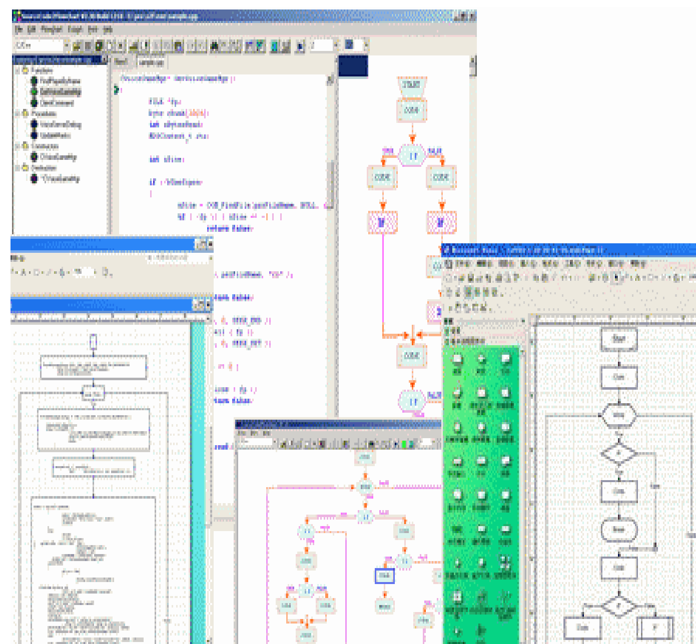





# Design Recovery Tools (2)

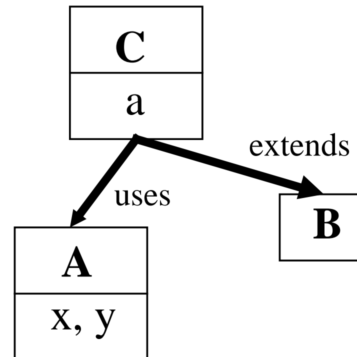
**CVF 3.0** is a automated program Flow chart generator. It can perform automated reverse engineering of program code into **programming flowcharts**, help programmers to document, visualize and understand source code.

It works with: C, C++, VC++, VB, VBA, VBScript, ASP, Visual C#, Visual Basic .NET, Visual J# .NET, VC++.NET, ASP.NET, Java, JSP, JavaScript, Delphi, PowerBuilder and Perl.



# UML Class Diagram Recovery

```
Class A {
   int x, y
   …
}
Class B {
   …
}
Class C extend B {
    A a
   …
}
```

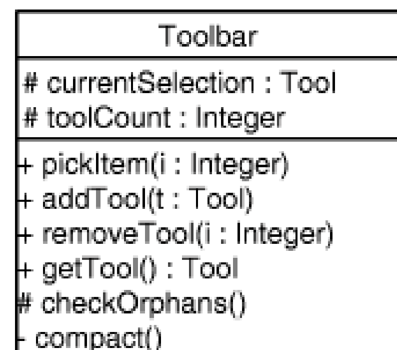





# Information shown (1)

- **Class property**
  - **- fields**
  - **- methods**
- **Element property**
  - **- types**
  - **- visibility**

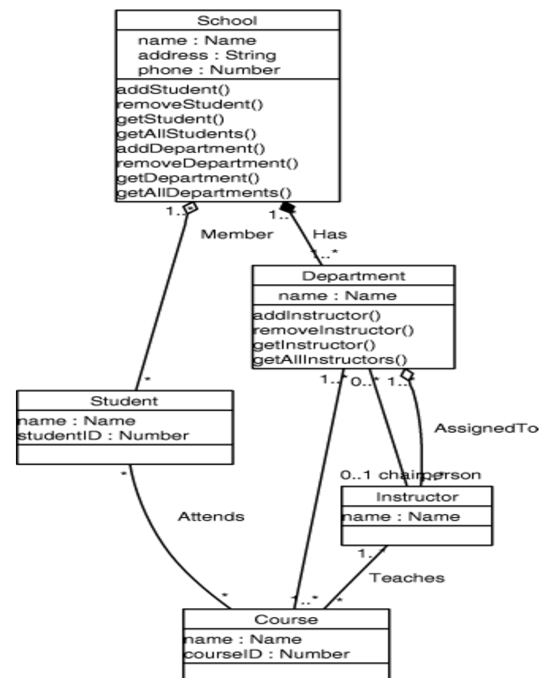
```
class Toolbar {
    protected Tool currentSelection;
    protected Integer toolCount;
    public void pickItem(Integer i) {}
    public void addTool(Tool t) {}
    public void removeTool(Integer i) {}
    public Tool getTool() {}
    protected void checkOrphans() {}
    private void compact() {}
}
```



# Information shown (2)

Relationships

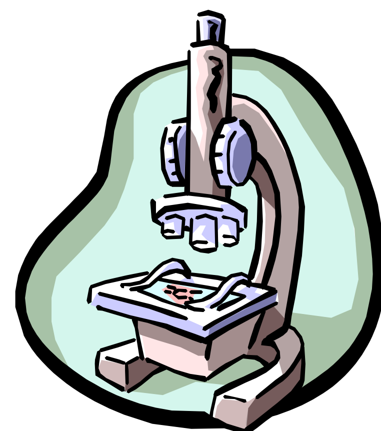
- **Inheritance/realization**: a class extends/implements a class/interface.
- **Aggregation/composition**: a class is part of another class.
- **Association**: a class holds a stable reference toward another class.
- **Dependency**: a change in a class might impact another class.





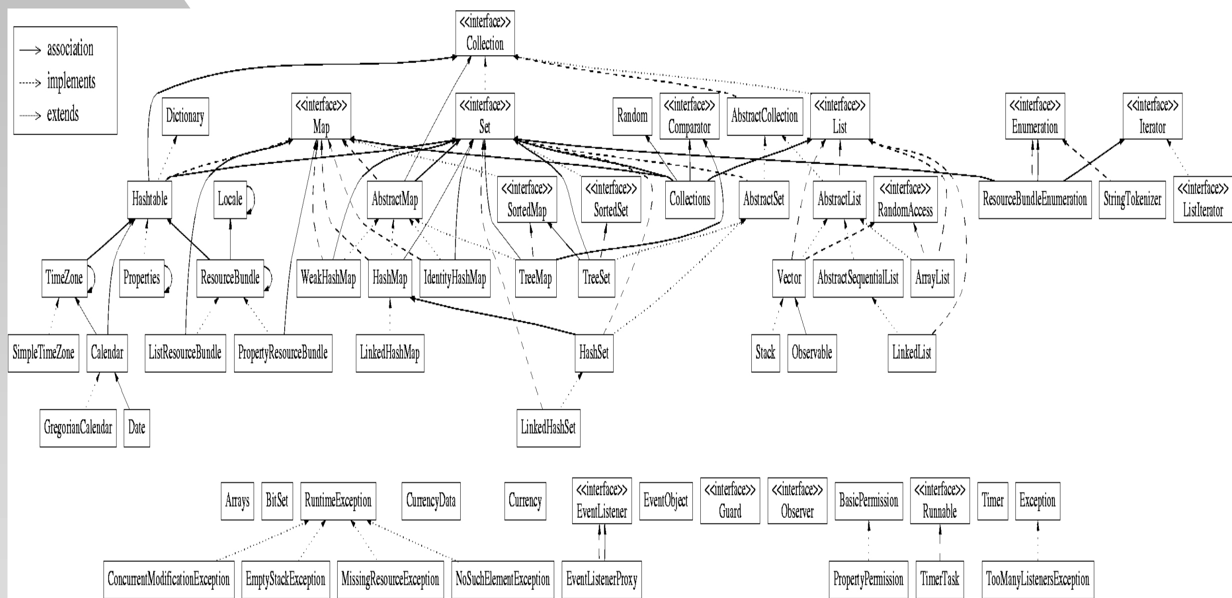

# Limits of UML Class Diagram Recovery Tools

1. **For a medium size system (the order of 20K LOC) it is quite common to have 50-100 classes. A design diagram reverse engineered from the code that shows them, even without displayning any property, is completely unreadable for human, whose cognitive abilities permit grasping information related to 5-10 objects at most.**
2. **Not all the useful information can be recovered (statically) from the code.**



# Example

**Java.util** (14 KLOC, 41 classes and 13 interfaces)





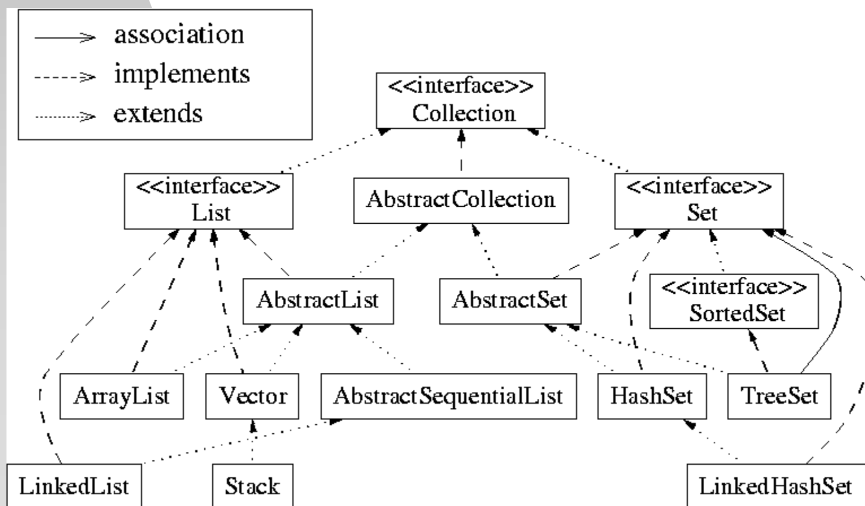

# Information missing

- Once implemented **aggregation** and **association** are indistinguishable.
- **Multiplicity** can not be recovered (statically is undecidible to determine the number of objects involved in a given relationship).
- Impossibility to recover **relation name and roles.**
- When **weakly typed containers** are used, the actual class of the contained objects is not known.
- Tagged values, constraints, properties and comment notes cannot be recovered automatically.

# Filtering

- **By filtering, users specify which information is irrelevant and can be skipped.**

**Java.util (only List and Set, No attributes)**



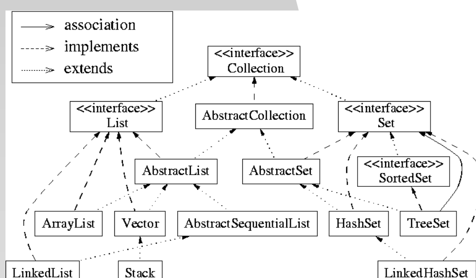



# Multiple Views

- **When defining multiple views for a given system, programmers decide which elements (classes, fields, methods, …) belong to which view.**

**Java.util**

**VIEW1: only List and Set** **VIEW2: …** **VIEW3: …**

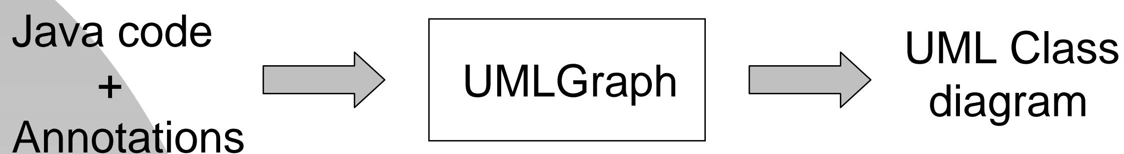


# Code annotations

- **Code annotations** are introduced to overcome the limitation of reverse engineering described above and to refine the default display options into more useful ones.
- Code annotations “guide” reverse engineering tools.
- **Code annotations** allows Filtering and Multiple views.
- They respect the **Javadoc syntax** of annotations (@ preceds the name of the annotation).



# UMLGraph (http:/www.spinellis.gr/sw/umlgraph/)




- The UML Class diagram is generated in Graphviz format that can be easly transformed in Postcript, Gif, Jpeg, …

Example:

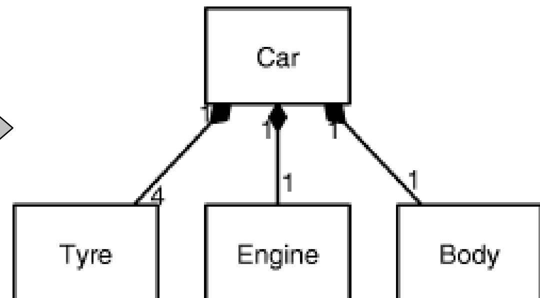
**javadoc -docletpath UmlGraph.jar -doclet UmlGraph -private Simple.java**

*javadoc will create by default a file named graph.dot in the current directory*

**dot -Tps -ograph.ps graph.dot**

# First example

```
class Tyre {}
class Engine {}
class Body {}
/**
 * @composed 1 - 4 Tyre
 * @composed 1 - 1 Engine
 * @composed 1 - 1 Body
 */
class Car {}
```






# Modelling

The **UMLGraph** class diagrams allows you to model:

- classes (specified as Java classes)
- attributes (specified as Java class fields)
- operations (specified as Java class methods)
- implementation relationships (specified using the Java implements declaration)
- generalization relationships (specified using the Java extends declaration or (for multiple inheritance) the javadoc **@extends** tag)
- association relationships (**@assoc** tag)
- navigatable association relationships (**@navassoc** tag)
- aggregation relationships (**@has** tag)
- composition relationships (**@composed** tag)
- dependency relationships (**@depend** tag)

# Relationship

All relationship tags appart from @extends take four arguments:

- The source adornments (role, multiplicity, and visibility)
- The relationship name
- The target adornments (role, multiplicity, and visibility)
- The target class

Example:

**/** @navassoc "1\n\n+owner\r" - "*\n\n+key" Password */**

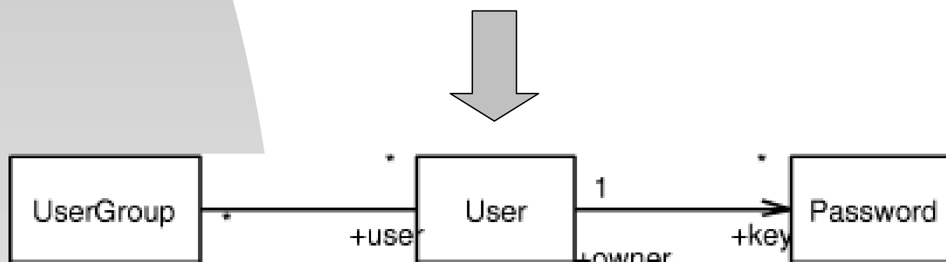


# Relationship example

```
/** @assoc * - "*\n\n+user " User */
class UserGroup {}

/** @navassoc "1\n\n+owner\r" - "*\n\n+key" Password */
class User{}

class Password{}
```



# Global vs Local setting

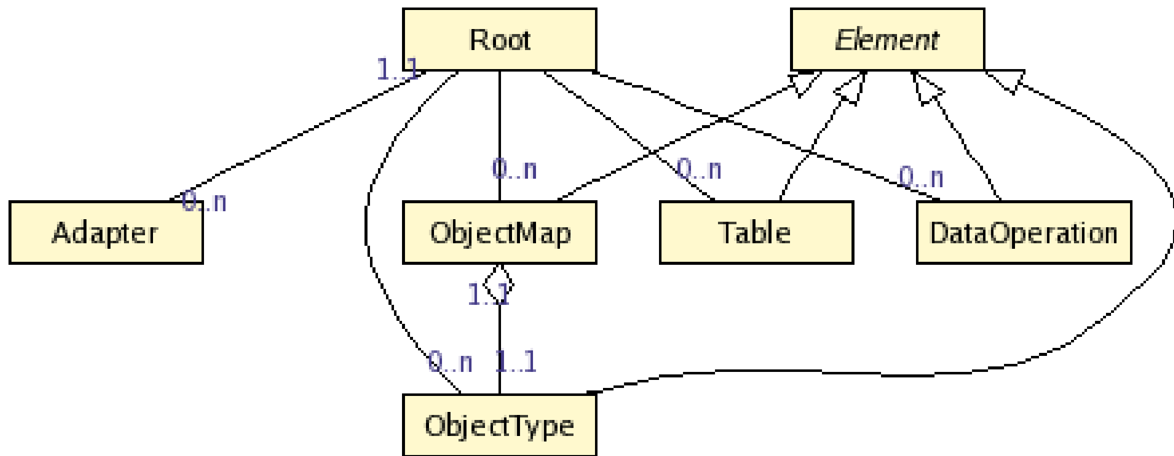
- Programmers can decide to change the default show/hide setting for classes/interfaces in two ways: **Global level** or at the **Local level.**
- **Default setting:** only class name (no fields, no methods, no associations, yes inheritance and realizations ).
- Global annotations change the setting **for all** classes of the project.
- Local (class) annotations override the global setting for the class being processed.



```
**
 * @assoc "1..1" - "0..n" Adapter
 * @assoc "" - "0..n" ObjectType
 * @assoc "" - "0..n" ObjectMap
 * @assoc "" - "0..n" Table
 * @assoc "" - "0..n" DataOperation
 **/
class Root {
   private Map m_adapters;
   private List m_types;
   private List m_maps;
   private List m_tables;
   private List m_ops;
   public Adapter getAdapter(Class klass) {}
}
```

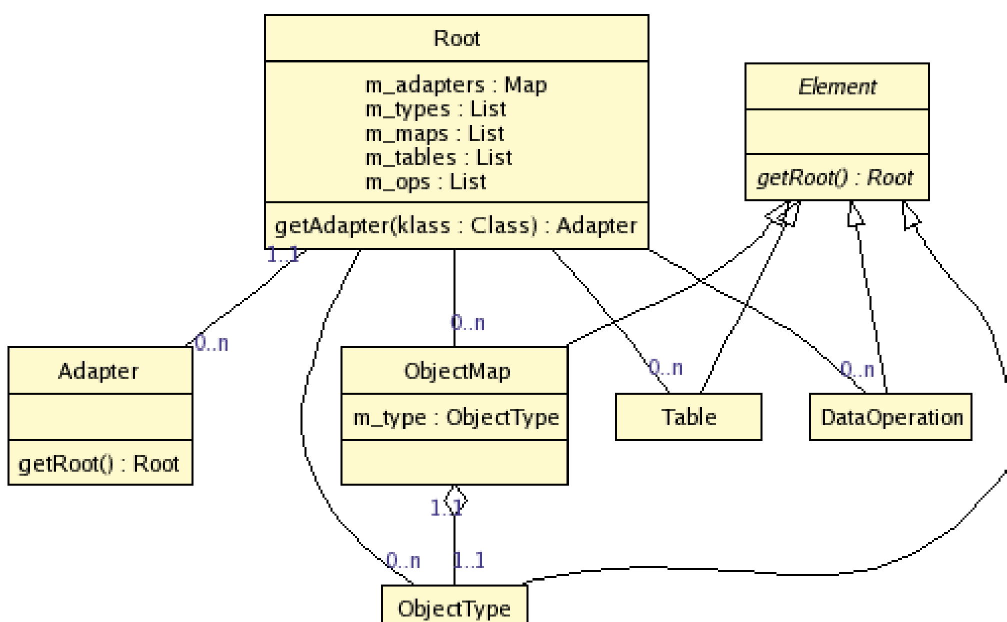
```
class Adapter {
   public Root getRoot();
}
abstract class Element {
   Root getRoot() {}
}
class ObjectType extends Element {}
/**
 * @has "1..1" - "1..1" ObjectType
 **/
class ObjectMap extends Element {
   private ObjectType m_type;
}
class Table extends Element {}
class DataOperation extends Element {}
```

# Default diagram






# Adding Fields, Methods and Types



# Global annotations

- Global annotations are specified in front of a special class named *UMLOptions (to add at the project).*

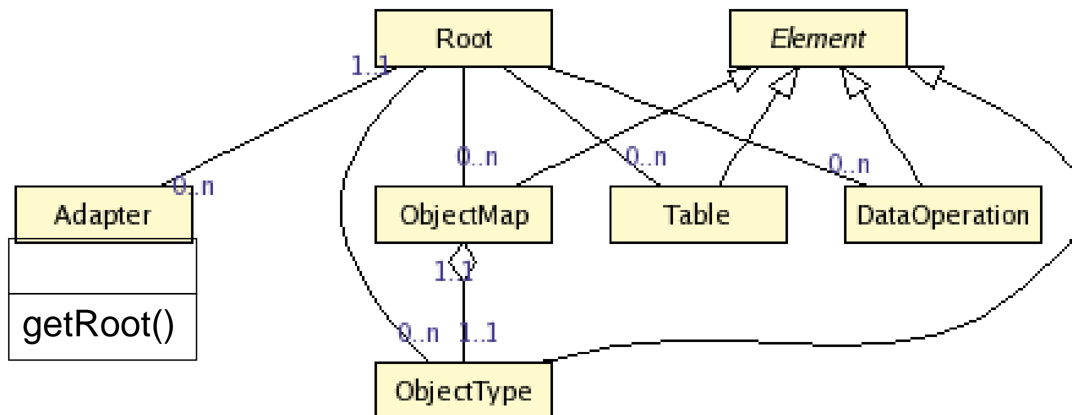
```
/**
 * @opt attributes
 * @opt operations
*  @opt types
 * @hidden
 */
class UMLOptions {}
```



# Local annotations

- Local annotations are specified in front of the class that we have to change.

```
/**
 * @opt attributes
 * @opt operations
 */
class Adapter {
   public Root getRoot();
}
```





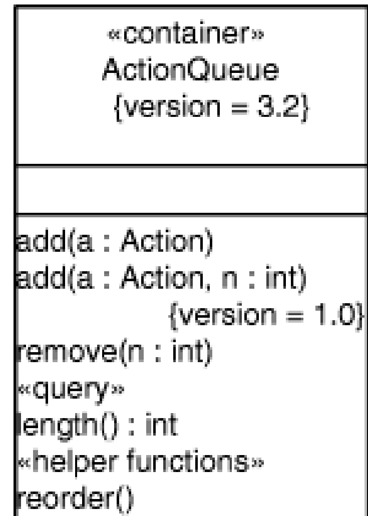

# Adding annotations to fields and methods

- Some annotations can be added directly to fields and methods:
  - **@hidden** hides a field or method
  - **@show** show a field or method
  - **@stereotype** add a stereotype
  - **@tagvalue** add a tagvalue

```
class Root {
    private Map m_adapters;
    private List m_types;
    /** @hidden */
    private List m_maps;
    private List m_tables;
    private List m_ops;
    /** @show */
    public Adapter getAdapter(Class klass) {}
}
```

# Stereotypes and tagged values

```
/**
 * @stereotype container
 * @tagvalue version 3.2
 */
class ActionQueue {
      void add(Action a) {};
      /** @tagvalue version 1.0 */
      void add(Action a, int n) {};
      void remove(int n) {};
      /** @stereotype query */
      int length() {};
      /** @stereotype "helper functions" */
      void reorder() {};
}
```






# New annotations "added"

- **@opt assoc_default**: show associations recovered automatically (when a field references an object of another class).

```
class A {
      B b;
}
```

- **@opt library**: show library classes.
- **@opt hide_all**: hide all classes.
- **@note:** add a note to a class.
- **@show:** show a class, an interface, a field or a method.

# @opt assoc_default

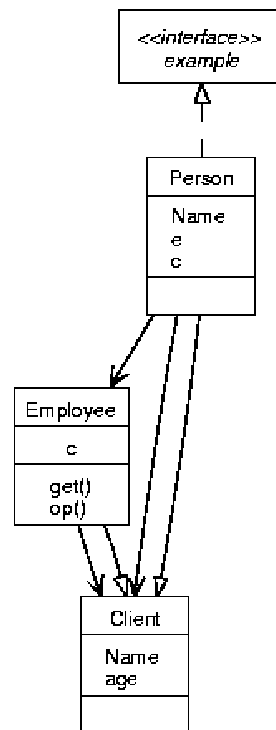
```
/**
* @hidden
* @opt operations
* @opt attributes
* @opt assoc_default
*/
class UMLOptions { }

interface example { }

/**
* @extends Client
*/
class Person implements example {
String Name;
Employee e;
Client c;
}

class Employee extends Client {
Client c;
Person get () { }
void op(Employee e) {Client c;}
}

class Client {
String Name;
int age;
}
```






# @note

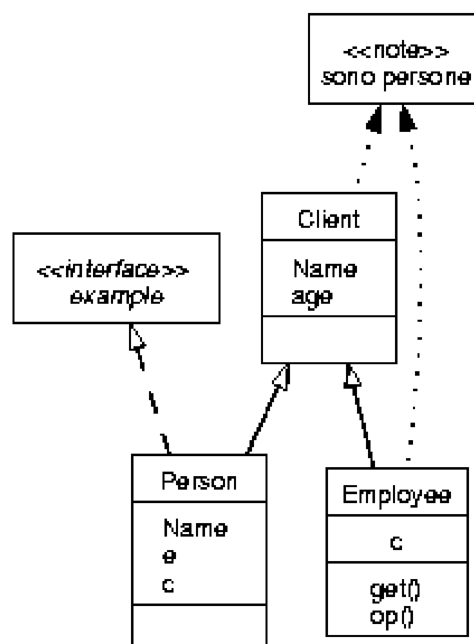
```
/**
*@hidden
*@opt operations
*@opt attributes
*/
class UMLOptions {}

interface example { }

/**
*@extends Client
*/
class Person implements example {
    String Name;
    Employee e;
    Client c;
}

/**
*@note "sono persone"
*/
class Employee extends Client {
    Client c;
    Person get () {}
    void op(Employee e) {Client c;}
}

/**
*@note "sono persone"
*/
class Client {
    String Name;
    int age;
}
```



# Views (1)

- To create a new view, it is sufficient to define a new class with a syntax similar to that of *UMLOptions.*

```
/**
 * @opt all
 * @hidden
 */
class View_1 {}
```

- To produce a view:

```
javadoc -docletpath UmlGraph.jar -doclet UmlGraph -view View_1 -private A.java
```



# Views (2)

- The setting of a single element (class, field, method, …) in a given view can be also changed, by adding an argument to the annotation that specifies the specifies the target view.

Example:

```
/**  @hidden view_1 */
Class student {
   …
 }
```

# View Definition Process

- The process for the definition of a new view is **incremental** and operates through **successive refinements.** It starts from an **empty view** and it adds elements (classes or relationships) or it refines already included elements until the displayed view is an accurate representation of the view's intent.



# Steps

Define a new view (select a meaningful name)

Make the view empty (hide_all)

Repeat the following steps until a satisfactory view is obtained:

- Select a class that contributes to the view and make it visible (if not yet such)

Select a subset of its attributes/methods that are meaningful for the view and make them visible

If necessary, add a note to explain the role of this class in the view

**OR:**

- Select a pair of classes displayed in the view such that a relationship relevant for the view exists between them;

If necessary, decorate the chosen relationship with name, roles and multiplicity

**DISPLAY THE NEW VIEW**

# UML Tools

- **UML diagram support**: supports "N" UML diagrams.
- **Forward engineering**: generates the source code of the classes with the methods stubbed out
- **Reverse engineering**
- **Round-trip**: synchronizes the UML models with the changes in the code
- Popular UML tools:
  - **Rational Rose**
  - **Together**
  - **Poseidon**
  - **Omondo**