A Use Case Template*

- Name:
  - [Uniquely Identifies this Use Case]
- Description:
  - [Brief description of user's goal]
- Actors:
  - [List of actors]
- Preconditions:
  - [Must be met before executing this use case]
- Main Sequence:
  - [Main sequence (i.e., scenario), with steps 1, 2, 3, …]
- Alternates:
  - [Variations of the main sequence]

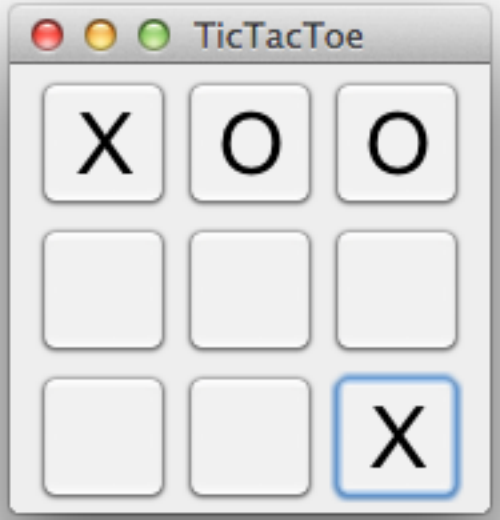
* Note: There exists no industry standard for Use Case descriptions!

Use Cases vs. User Stories

- User Stories describe what the customer needs
- Use Cases describe what the product does
  - Contain a substantial amount of detail
- Both Use Cases and User Stories are written in customer's business language

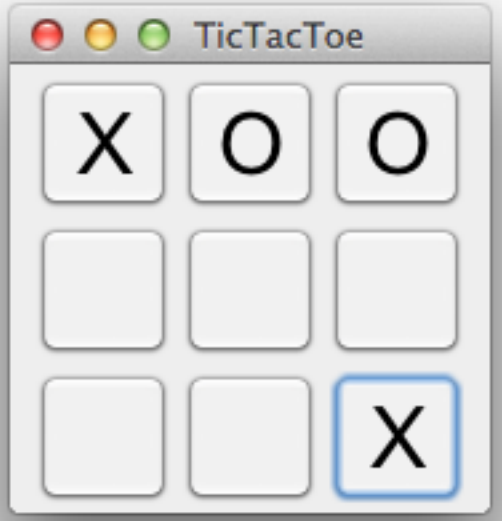
Use Case Scenarios Exercise

- How to Write a main scenario for a Player's normal move in TicTacToe?



Use Case Scenarios Exercise

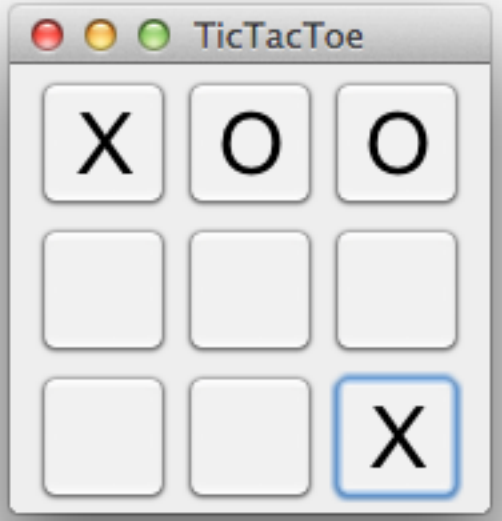
- How to Write a main scenario for a Player's normal move in TicTacToe?
  - Focus on "happy path":
    - Game is initialized and started
    - It is the expected player's turn
    - Choice of square is available / empty
    - Player's move will not end up in win/lose/draw



Use Case Scenarios Exercise

- How to Write a main scenario for a Player's normal move in TicTacToe?

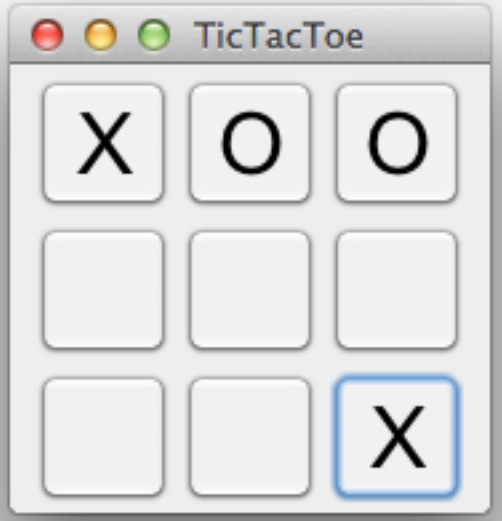
- What else do we need to consider?



Use Case Scenarios Exercise

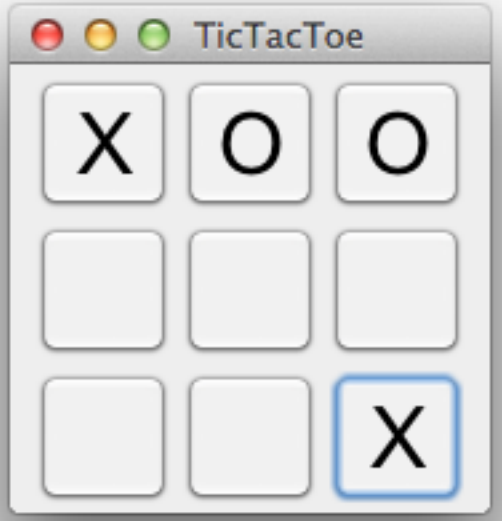
- How to Write a main scenario for a Player's normal move in TicTacToe?

- What else do we need to consider?
  - Write an alternate scenario for an illegal move
  - Write an alternate scenario for the end-of-game, Player wins



Use Case Scenarios Exercise

- How to Write a main scenario for a Player's normal move in TicTacToe?



- What else do we need to consider?
  - Write an alternate scenario for an illegal move
  - Write an alternate scenario for the end-of-game, Player wins
  - Write an alternate scenario for human vs. AI (computer), or this could be a main scenario…

Other ways of writing requirements

Other ways of writing requirements

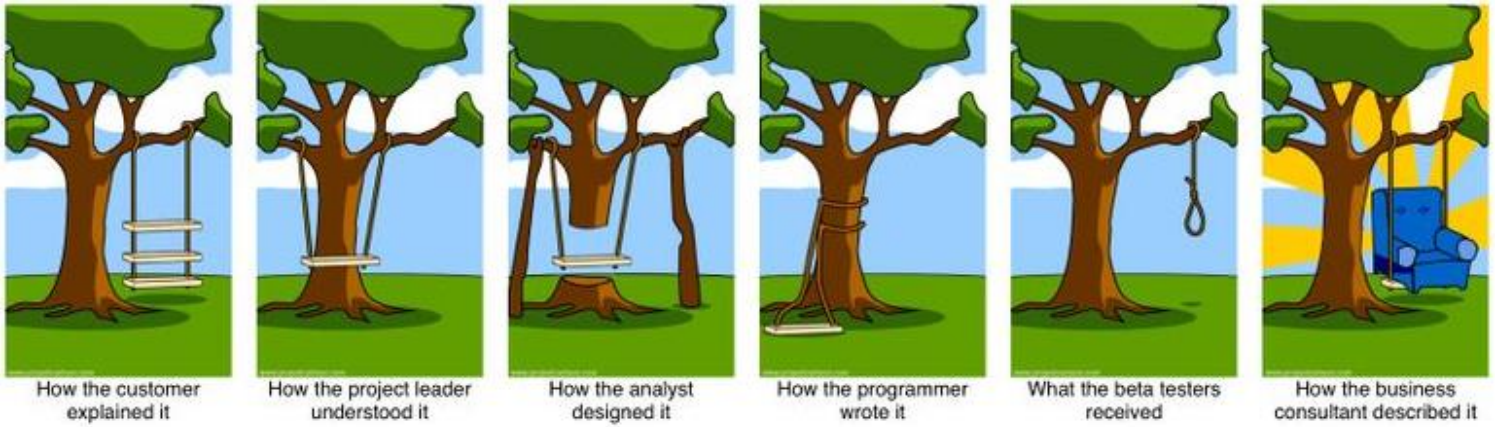
- Natural Language:
  - expressive, intuitive, universal ⇒ can be understood by stakeholders

Other ways of writing requirements

- Natural Language:
  - expressive, intuitive, universal ⇒ can be understood by stakeholders
  - ambiguous ⇒ leads to different interpretations

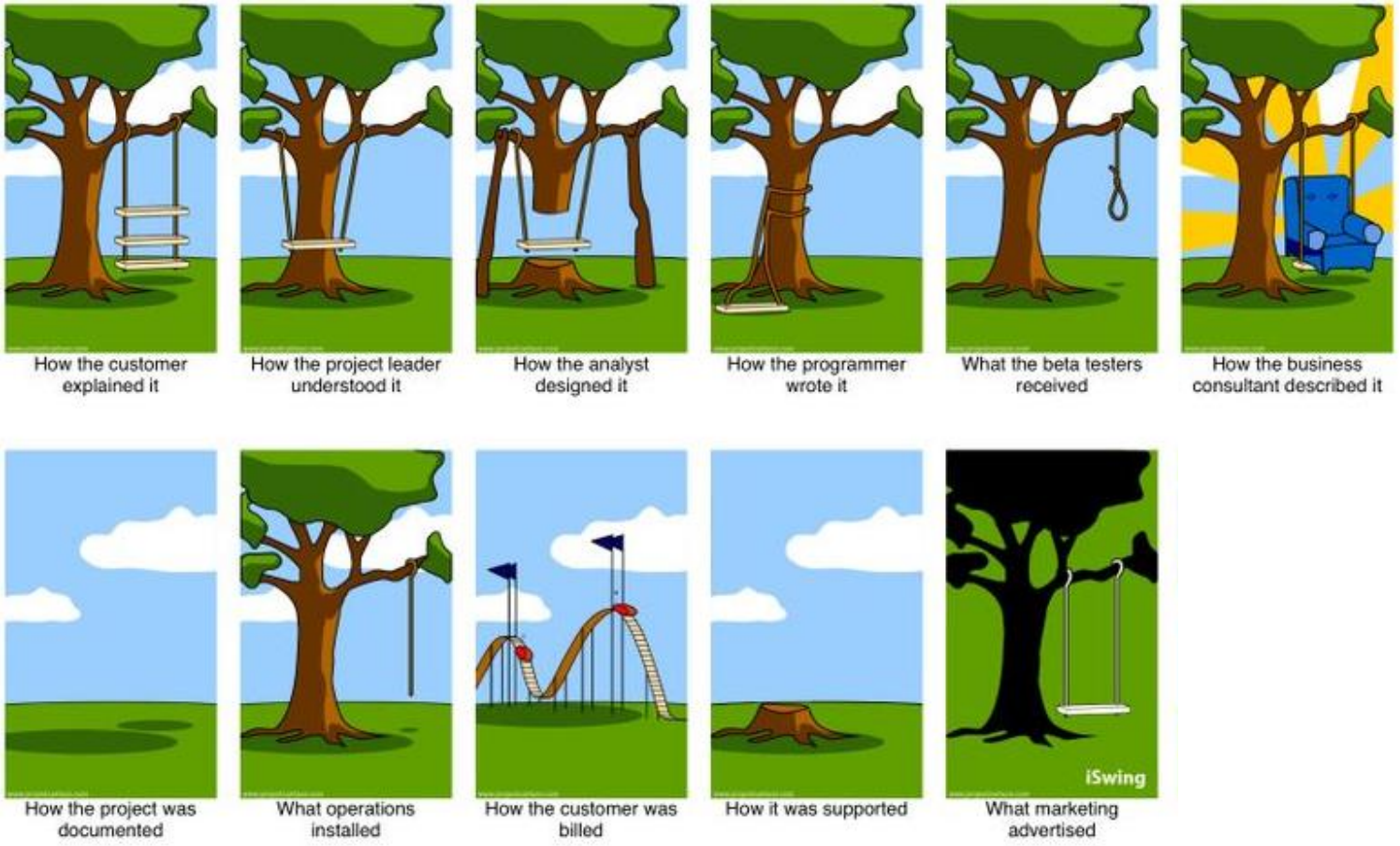
Other ways of writing requirements

- Natural Language:
  - expressive, intuitive, universal ⇒ can be understood by stakeholders
  - ambiguous ⇒ leads to different interpretations



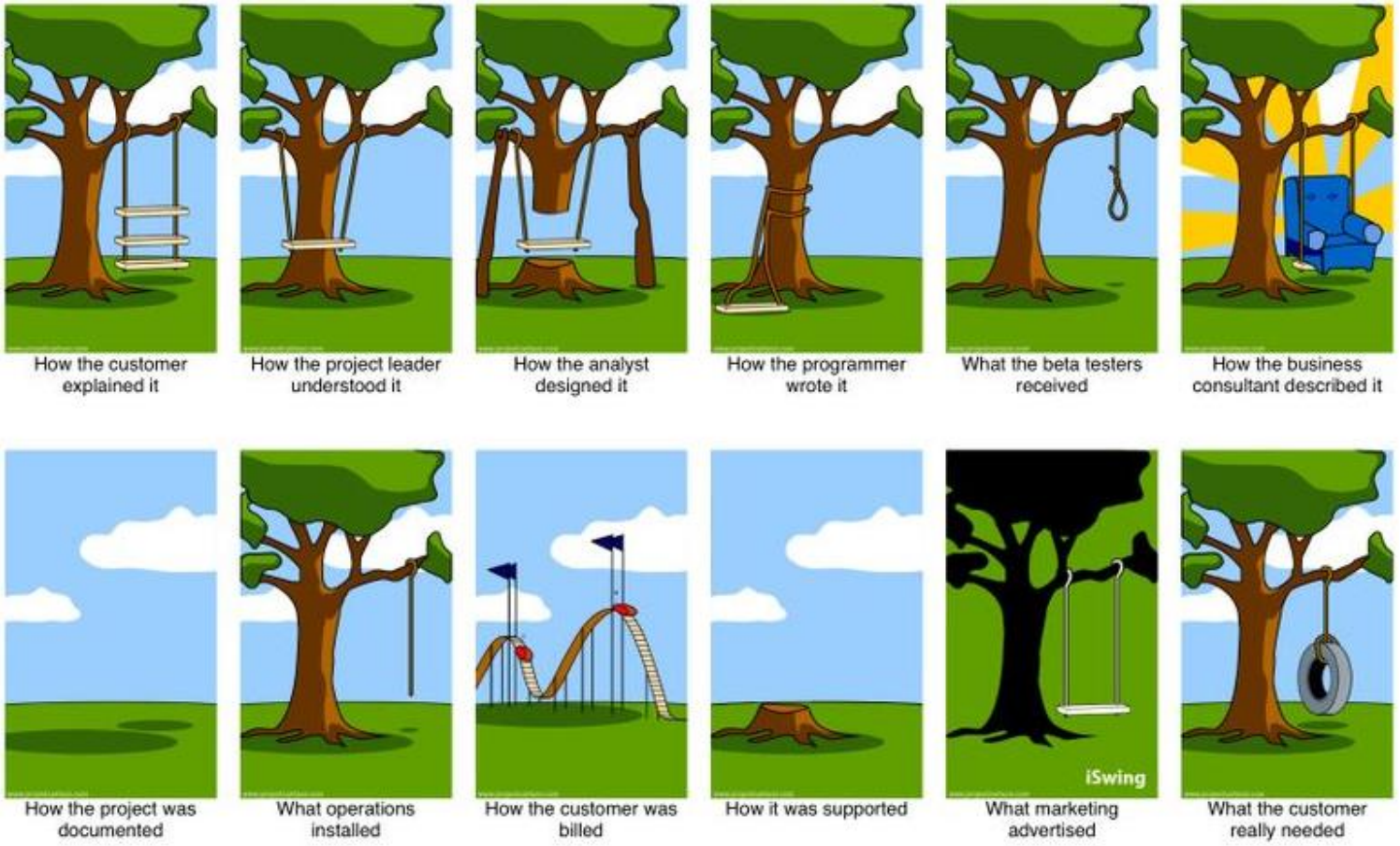
Other ways of writing requirements

- Natural Language:
  - expressive, intuitive, universal ⇒ can be understood by stakeholders
  - ambiguous ⇒ leads to different interpretations



Other ways of writing requirements

- Natural Language:
  - expressive, intuitive, universal ⇒ can be understood by stakeholders
  - ambiguous ⇒ leads to different interpretations



Other ways of writing requirements

- Natural Language:
  - expressive, intuitive, universal ⇒ can be understood by stakeholders
  - ambiguous ⇒ leads to different interpretations
- Structured Natural Language:
  - use standard form or template
- Mathematical Specifications
  - reduce ambiguity, but customers do not understand them
  - e.g., finite state machines, sets, etc.

Other ways of writing requirements

- Graphical notations:
  - graphical models, supplemented by text annotations
  - e.g., UML

After gathering requirements, and
before implementation,
we should … the software

After gathering requirements, and
before implementation,
we should design the software

UML = **U**nified **M**odeling **L**anguage

- http://uml.org/

- Introduced in the '90s
- Object-Oriented Programming was emerging
- Need to move to large / distributed software
- There was no "universal" way to document design of code ⇒ opportunity to create UML

UML = **U**nified **M**odeling **L**anguage

- general-purpose, developmental, modeling language used in software engineering

- is intended to provide a standard way to visualize the design of a system

UML Class Diagrams

Why UML Class Diagrams?

- To build a shared vision about how the software will work
- To provide a higher level of abstraction than source code, so team can focus on the most important details (i.e., "big picture")
- To document the implemented software for other people
  - New members of the team
  - Developers who will enhance/repair the software
  - People who bought the software from you on contract

Building a Shared Vision

- Different team members have different (or not) ideas about how the software will work

- Team needs to align everyone to get them working toward the same implementation (e.g., get everyone on the same page)

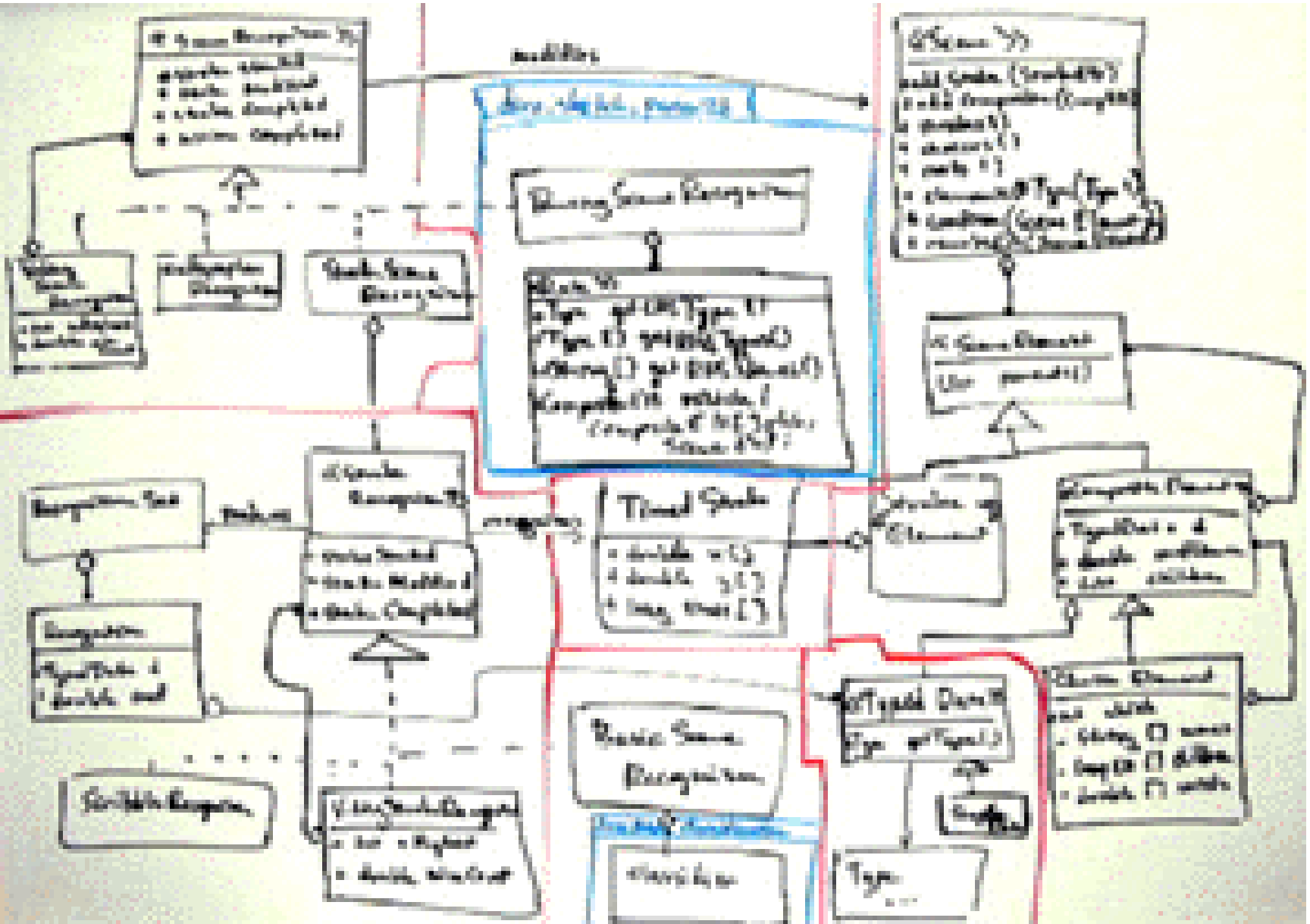
- This alignment often arises from UML diagrams sketched, championed and challenged on a whiteboard

How would you represent the following Source Code visually?

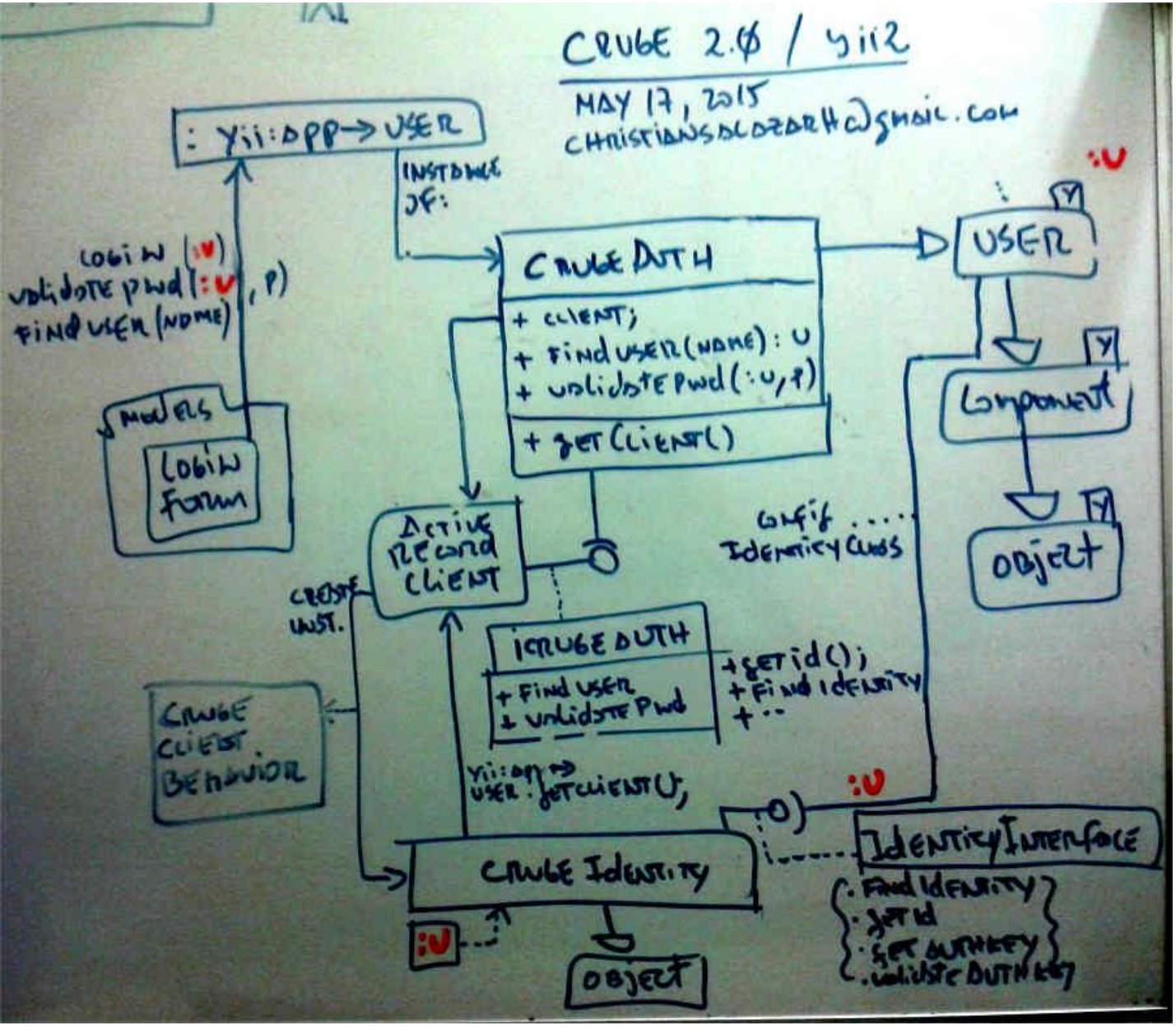
```
import java.util.Vector;

public class Driver {
    private StringContainer b = null;

    public static void main(String[] args){
        Driver d = new Driver();
        d.run();
    }

    public void run() {
        b = new StringContainer();
        b.add("One");
        b.add("Two");
        b.remove("One");
    }
}
```

To be continued…

UML Class Diagrams on Whiteboard



UML Class Diagrams on Whiteboard



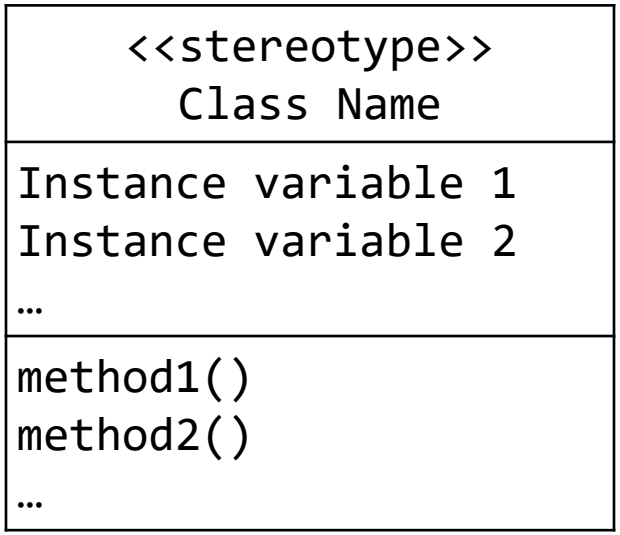
https://github.com/freesoftwarefactory/cruge/wiki

UML Class Diagrams for Creating Alignment

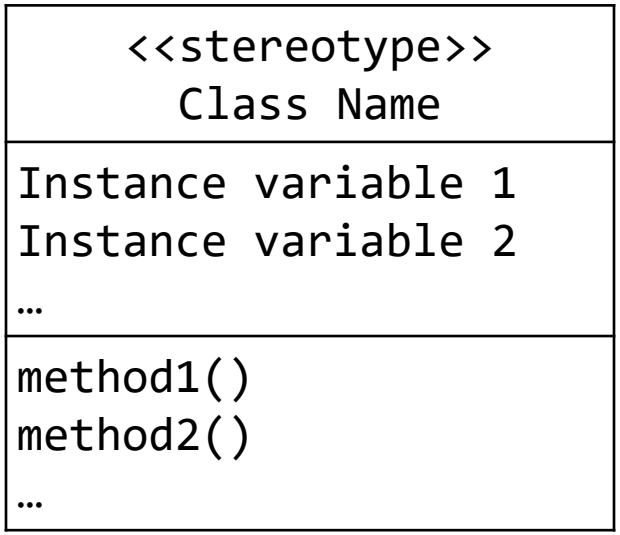
- Often focus on team member's agendas:
  - "It has to work like this because…"
- Tend to focus on contention ⇒ uncontested aspects of the implementation are not discussed
  - Private variables
  - Private members
- Tend to ignore completeness

UML Class Diagram
Brief Tutorial

How to represent a Class in a UML Class Diagram?



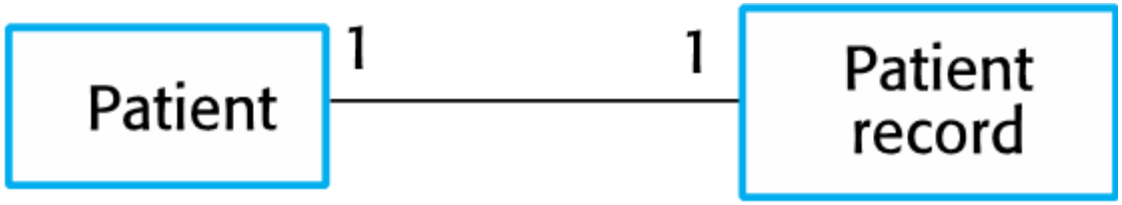
How to represent a Class in a UML Class Diagram?



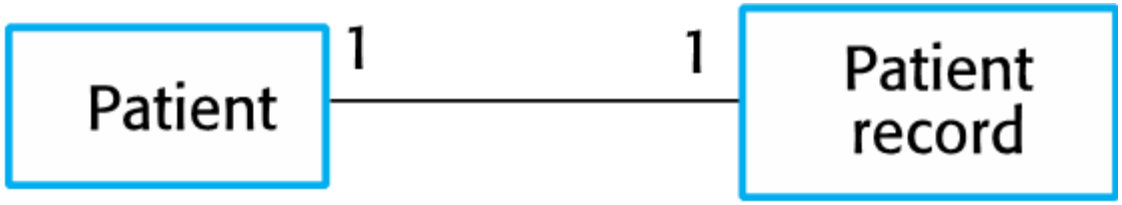
- Include
  - Class name
  - Public data (if any)
  - Public methods
  - Private data?
  - Private methods?
  - Data types?
  - Method parameters?

? = "if needed"

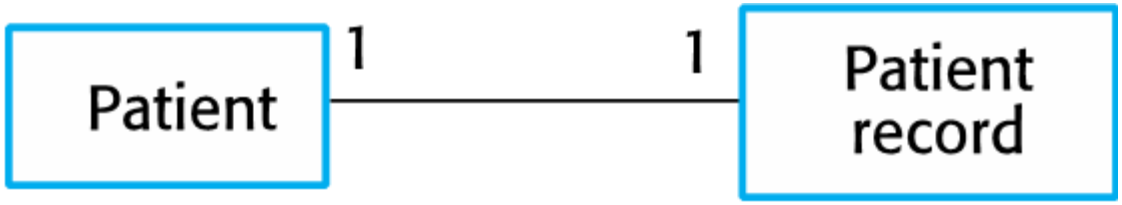
What type of relation is illustrated?



What type of relation is illustrated?

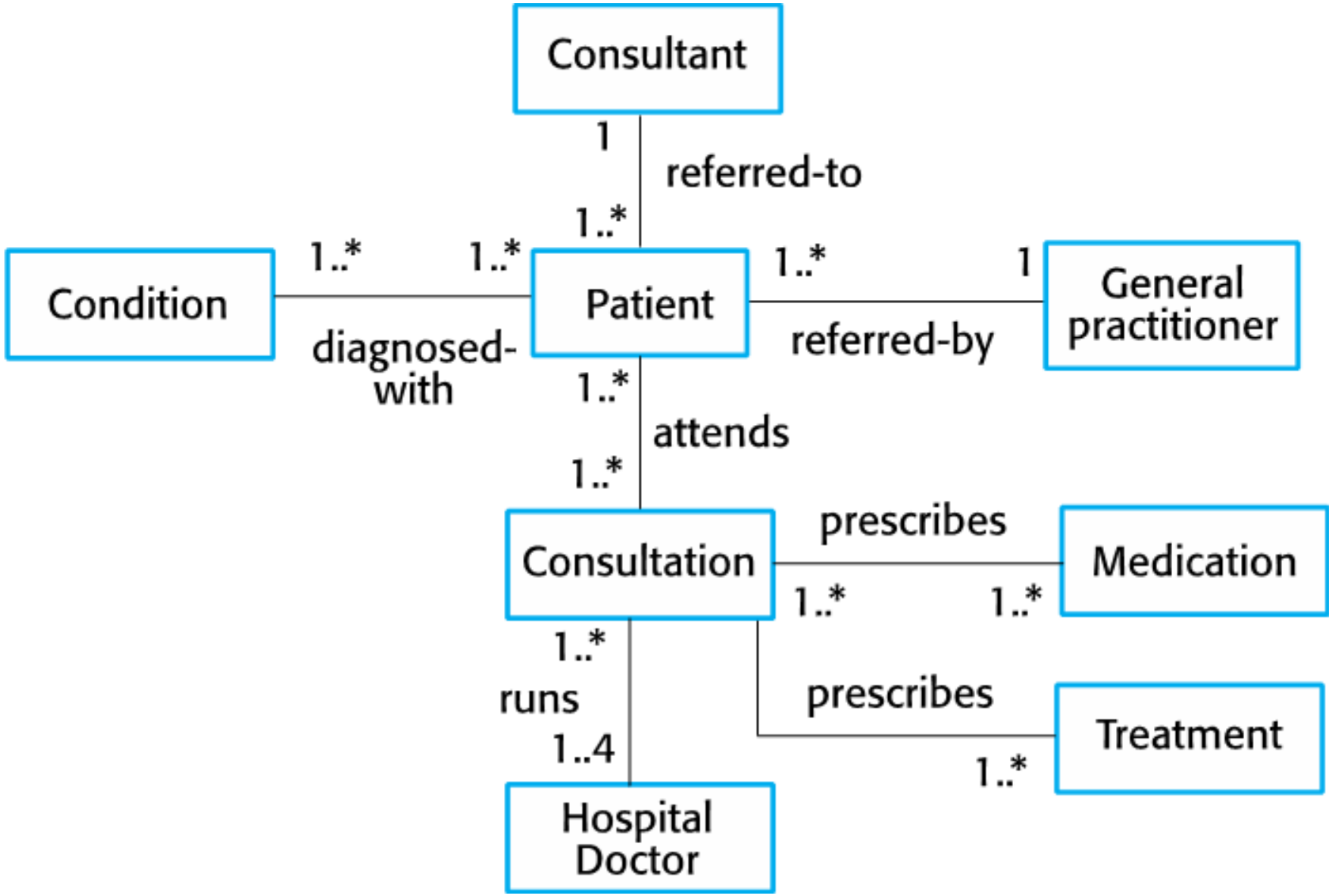


What type of relation is illustrated?

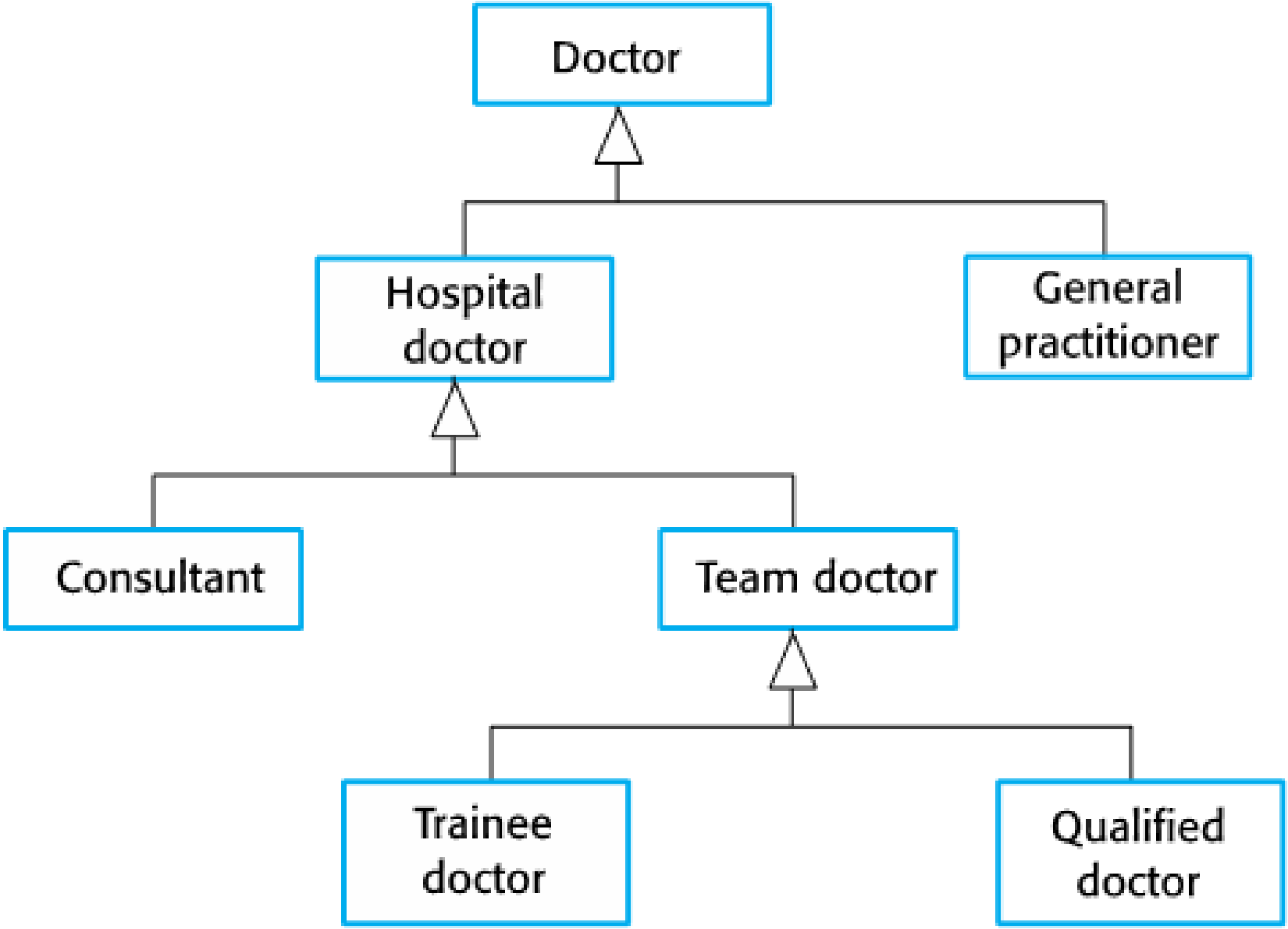


Association relation (very generic)

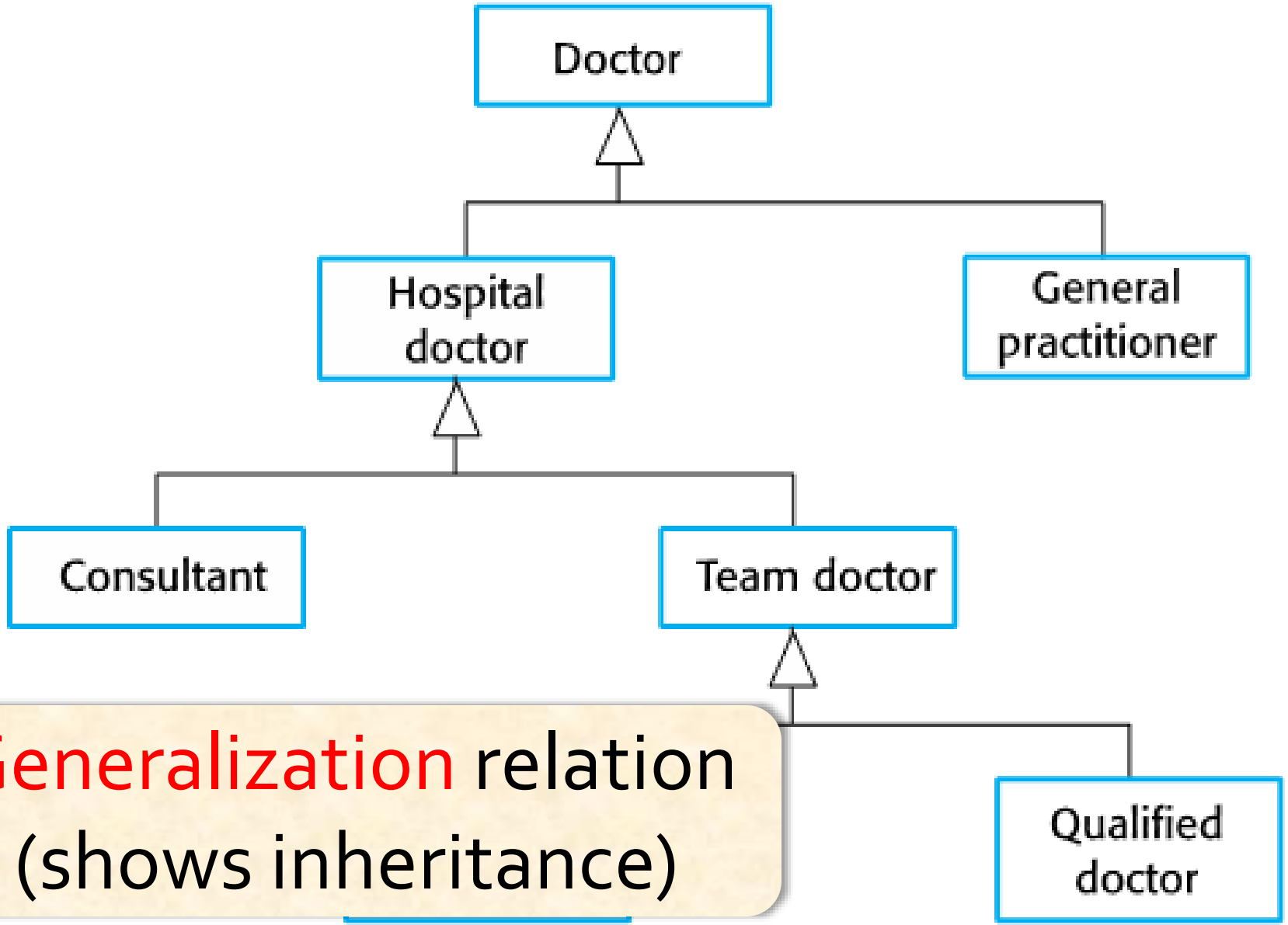
Association Example – Mentcare System



What type of relation is illustrated?

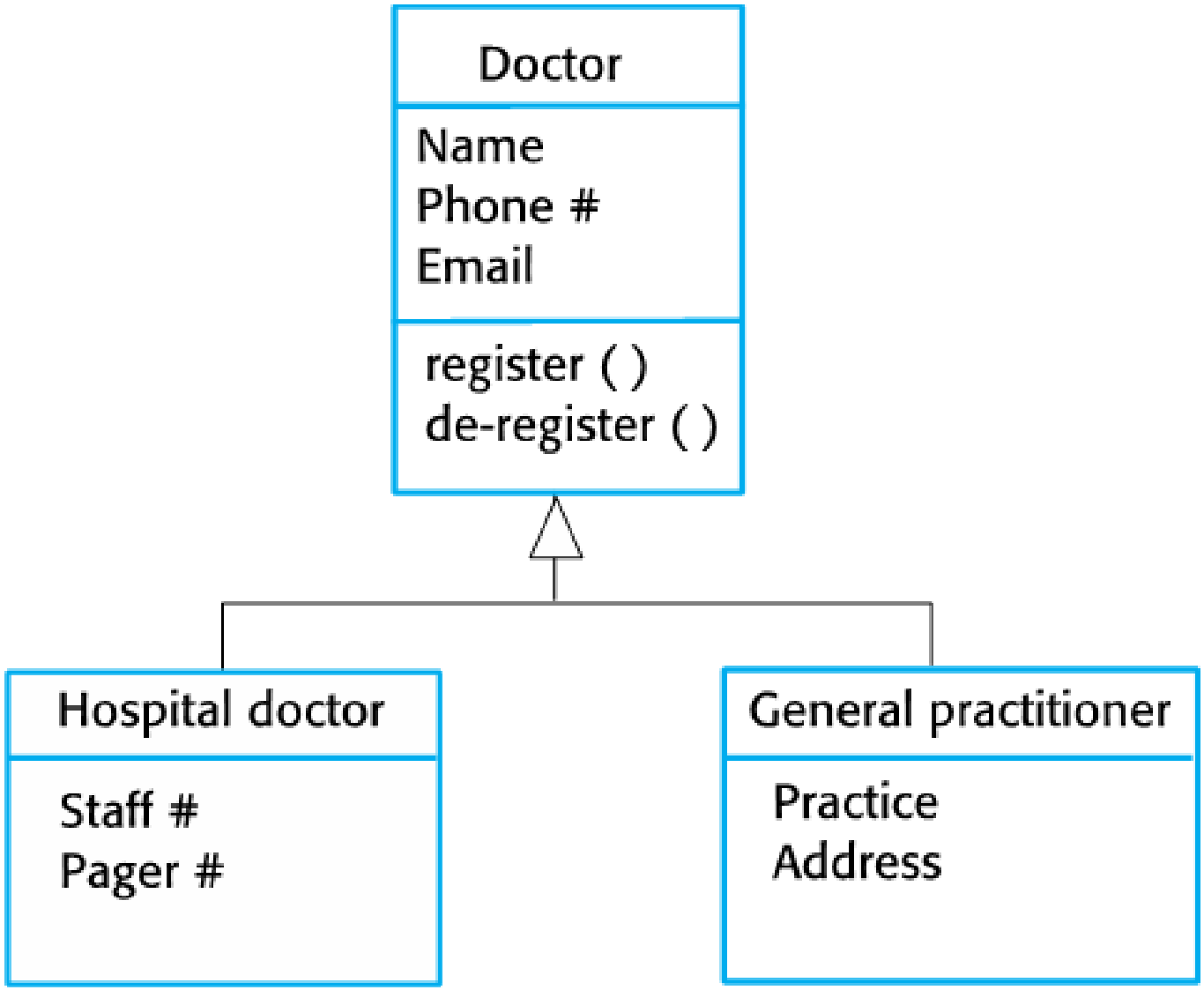


What type of relation is illustrated?

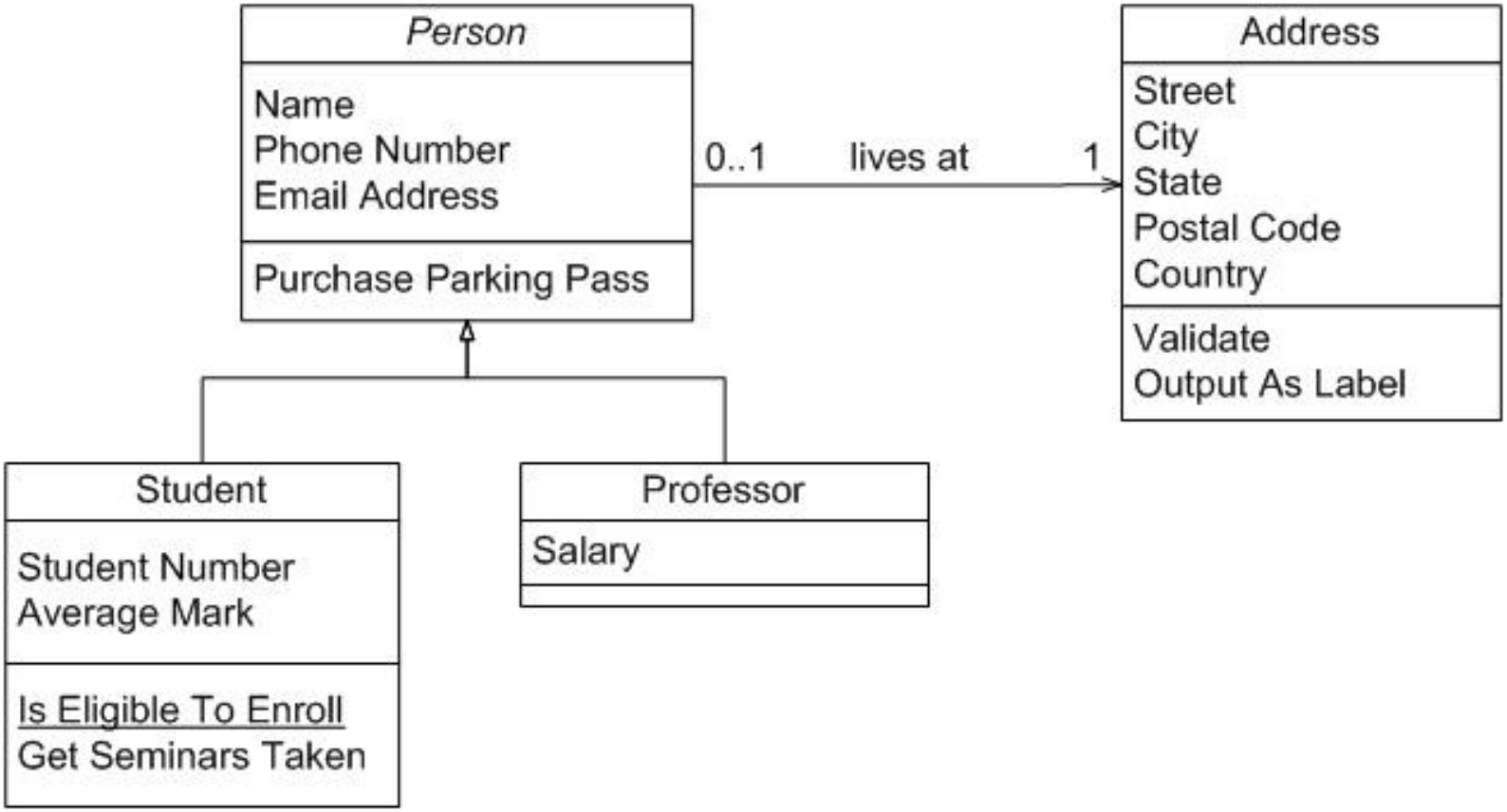


Generalization relation (shows inheritance)

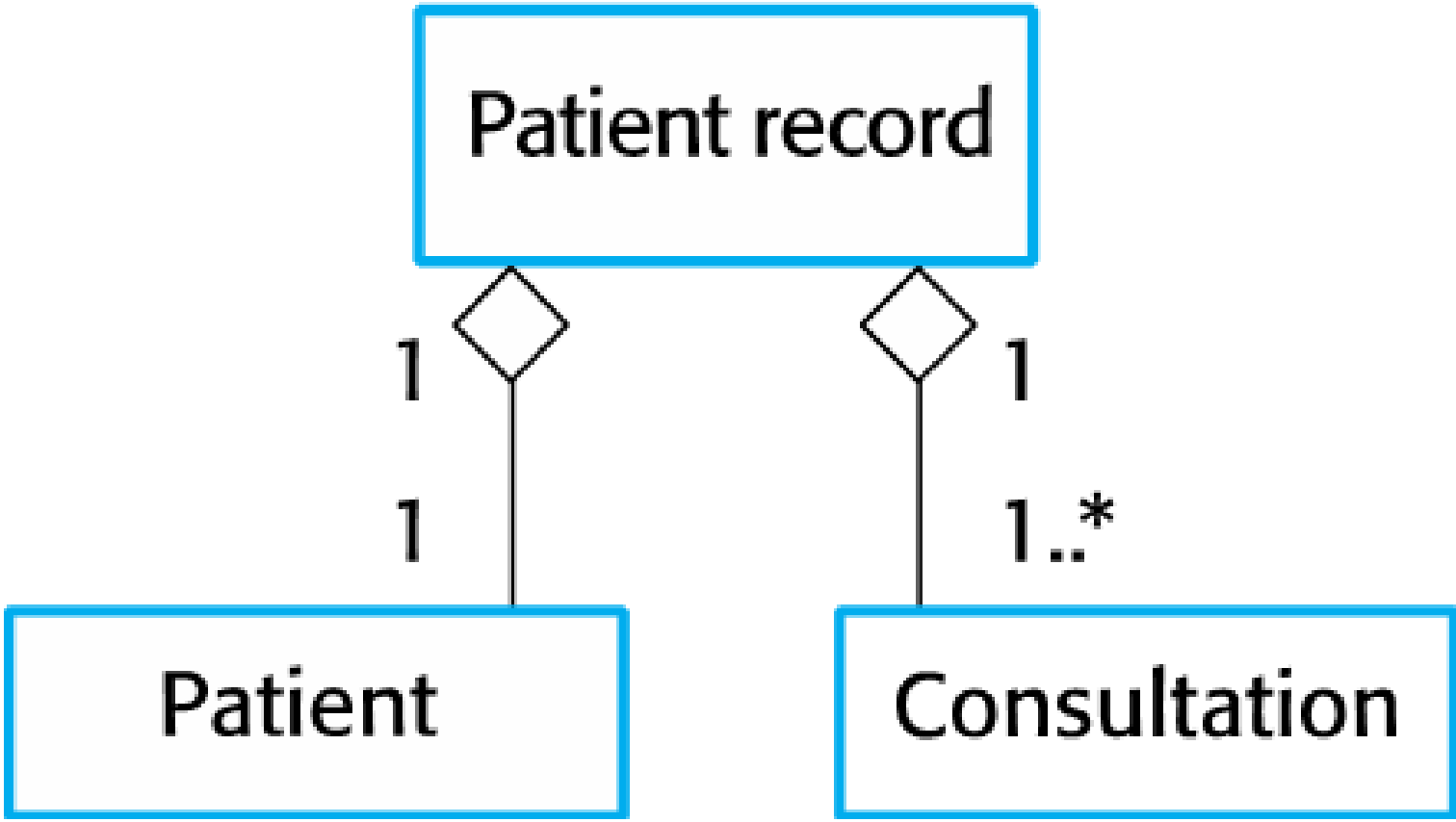
Example generalization hierarchy with added details



Example generalization and association



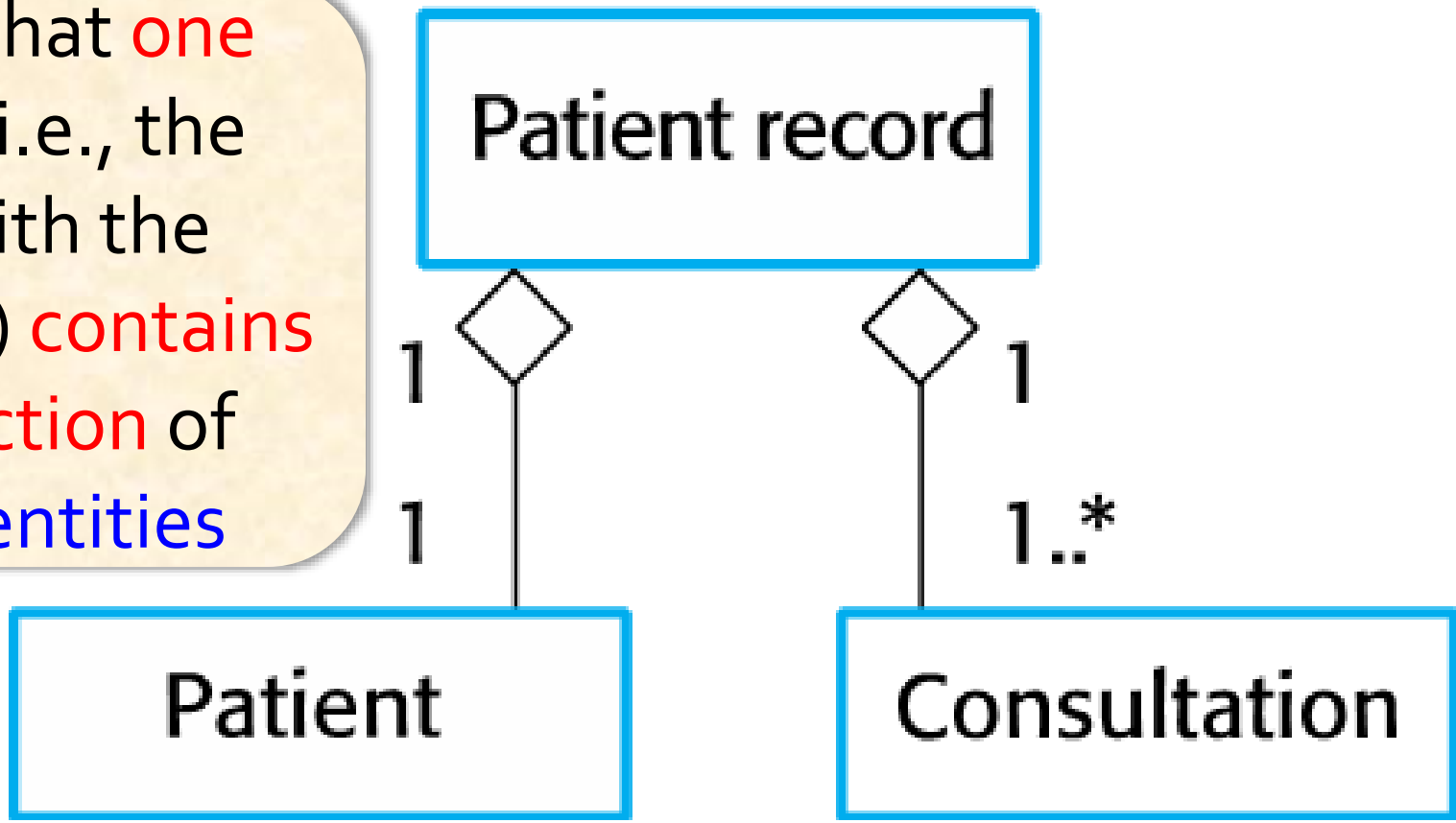
What type of relation is illustrated?



What type of relation is illustrated?

Aggregation relation

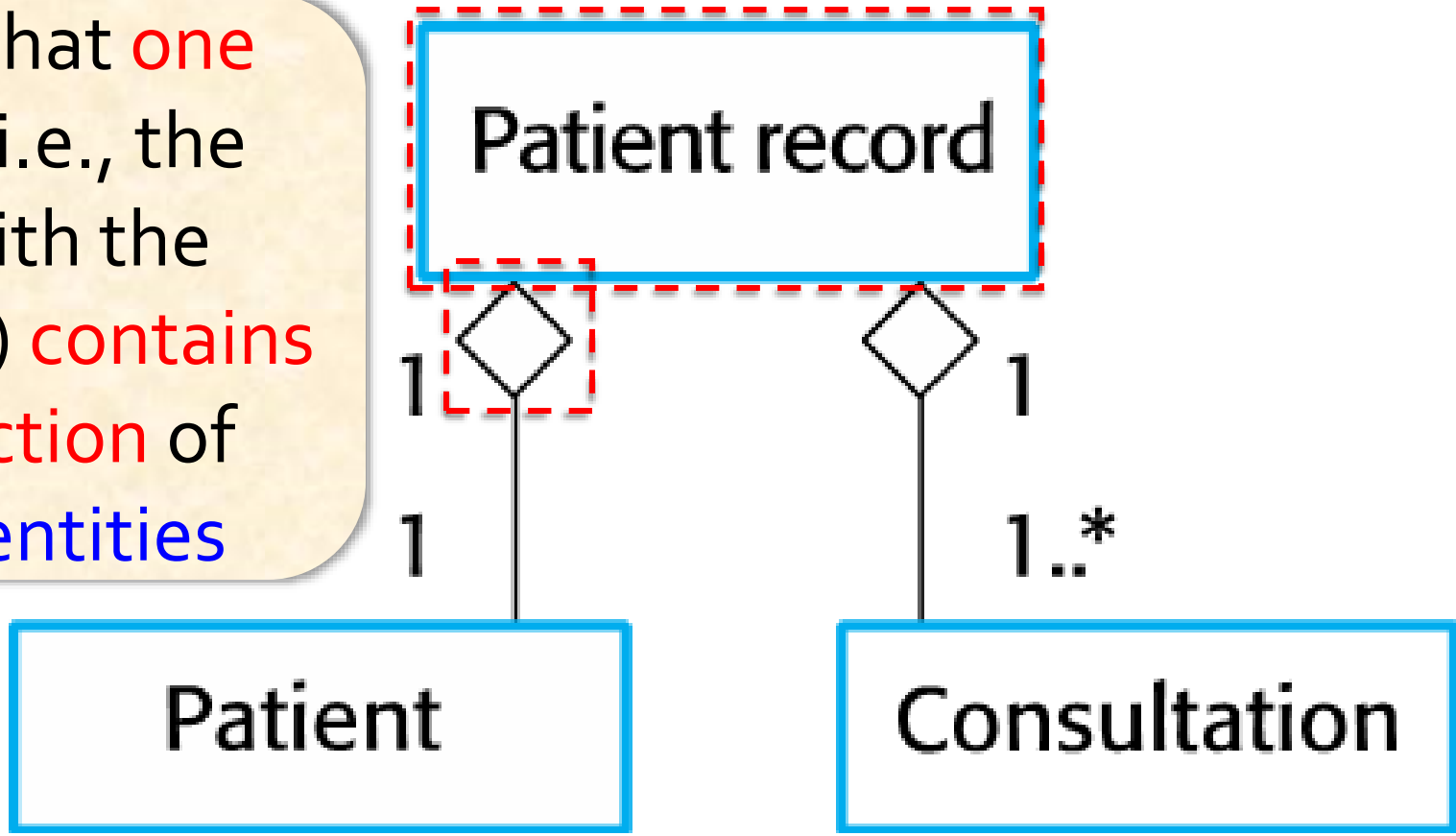
shows that one entity (i.e., the one with the diamond) contains a collection of other entities



What type of relation is illustrated?

Aggregation relation

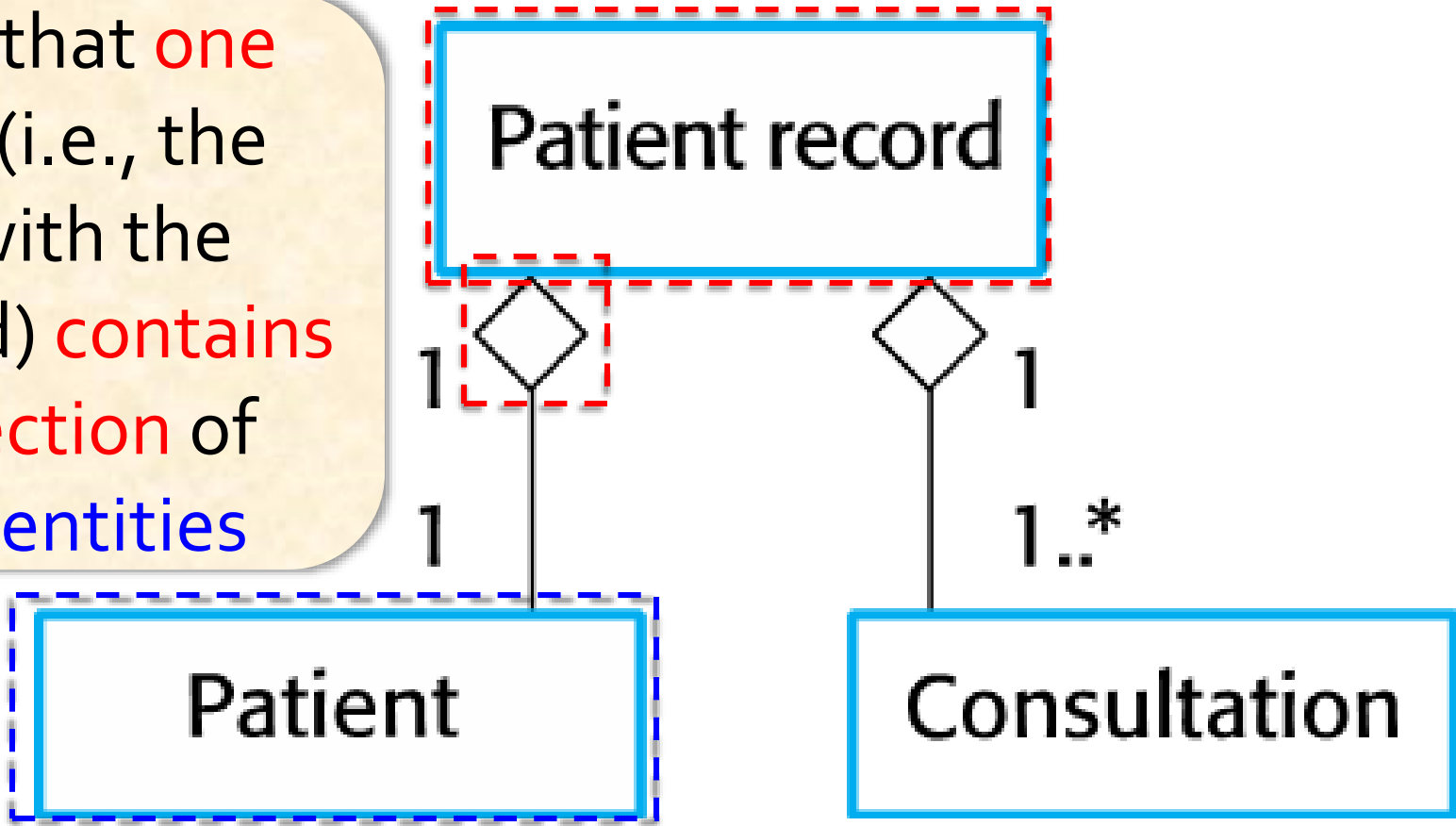
shows that one entity (i.e., the one with the diamond) contains a collection of other entities



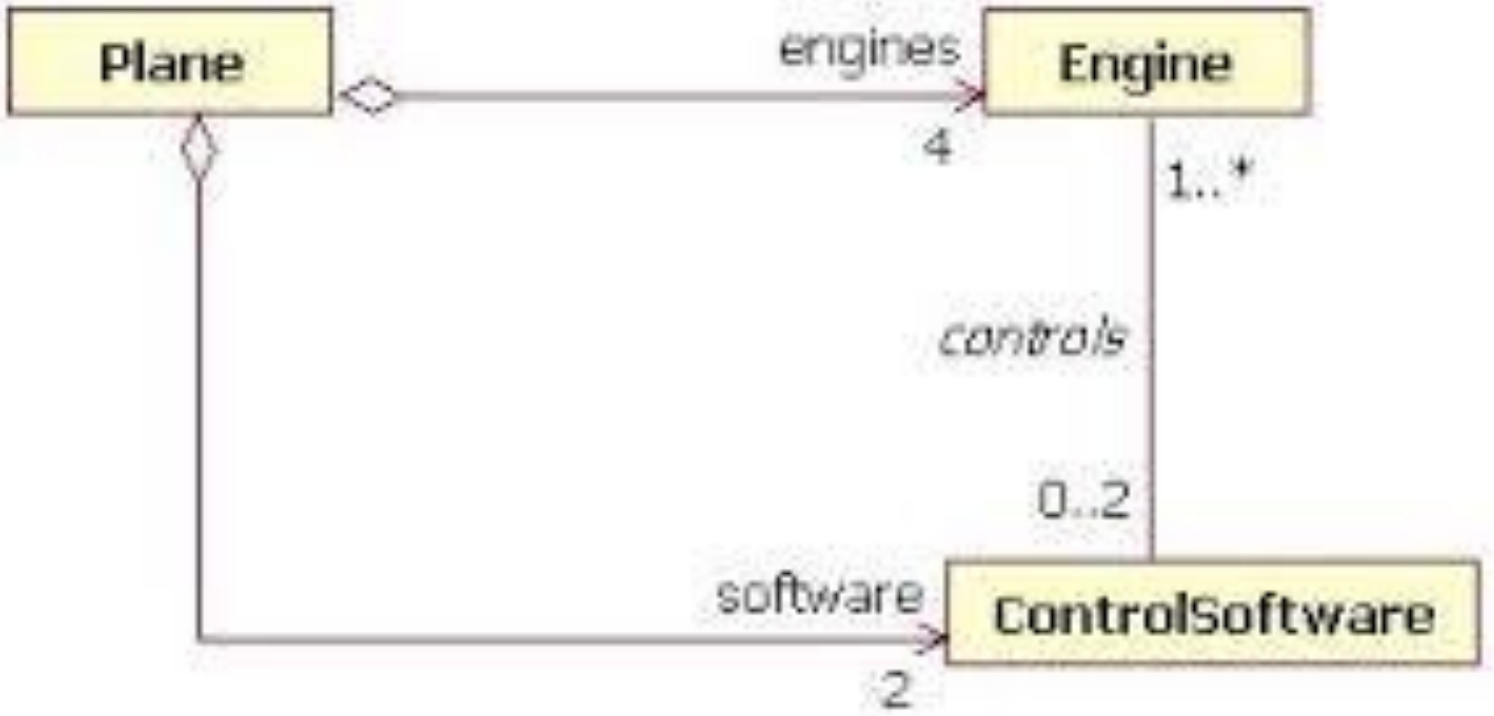
What type of relation is illustrated?

Aggregation relation

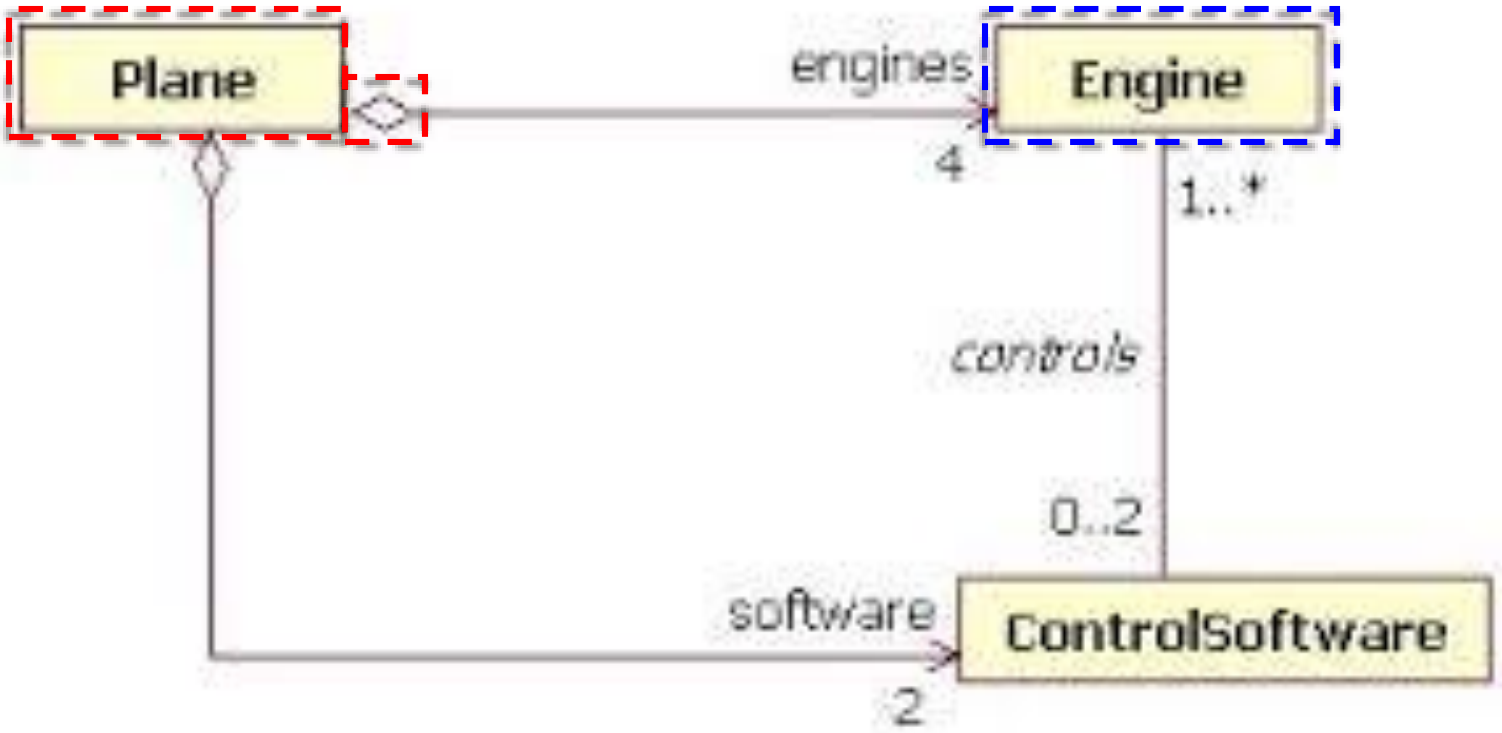
shows that one entity (i.e., the one with the diamond) contains a collection of other entities



Another Example Aggregation Relation



Another Example Aggregation Relation



Plane (i.e., the one with the diamond) contains a collection of Engine

Example of Composition Association

Composition
relation

Example of Composition Association

Composition relation

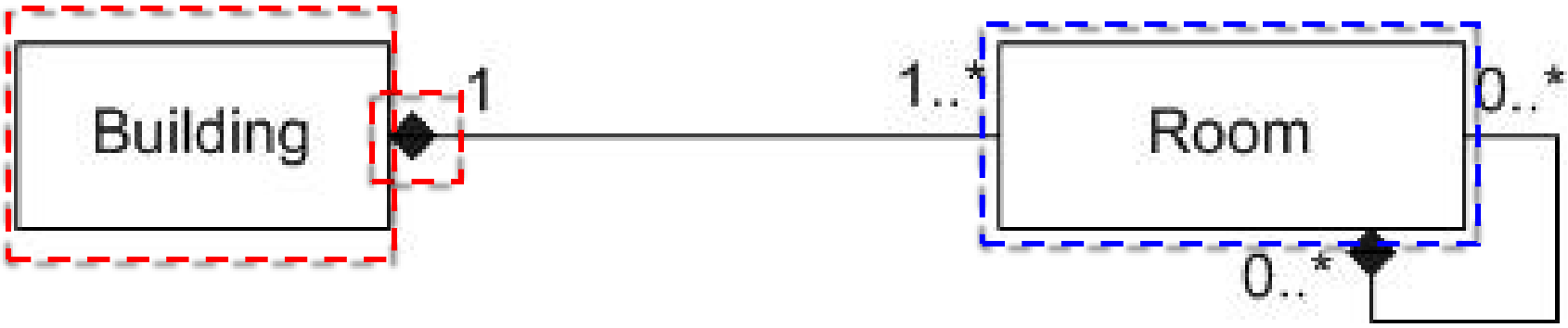
Example of Composition Association

Composition relation

shows that one entity (i.e., the one with the filled diamond) contains a collection of other entities, which the one entity "has more control over"

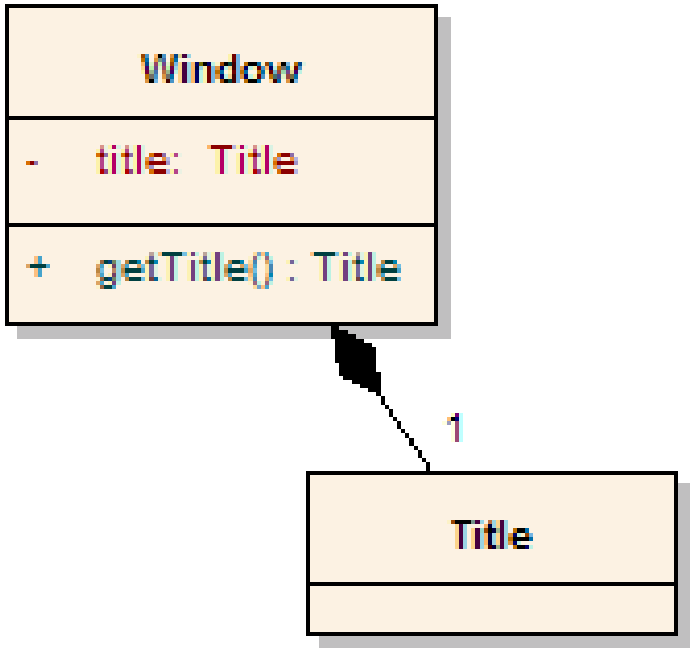
Example of Composition Association

Composition relation



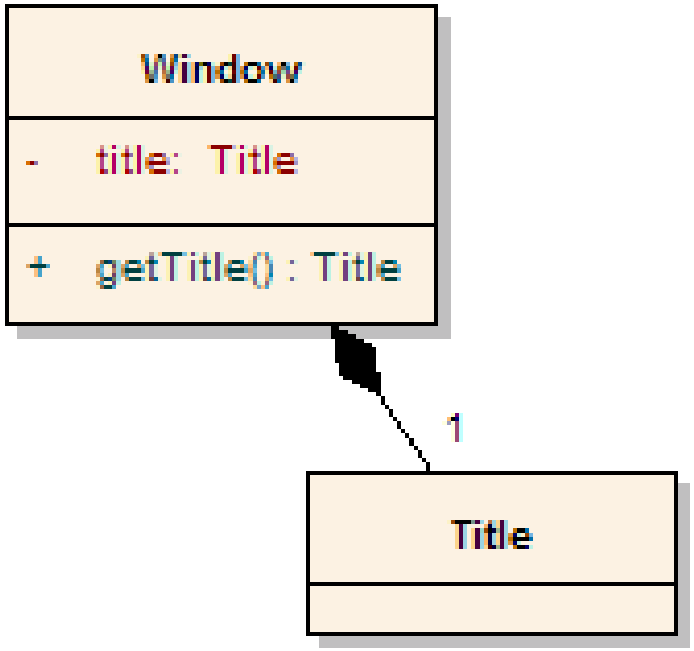
shows that one entity (i.e., the one with the filled diamond) contains a collection of other entities, which the one entity "has more control over"

Example of Composition Association




```
class Title {
        …
}

class Window {
        private $title;
        function __construct() {
                $this->title = new Title();
                $this->title->setText(…);
        }
}
```

Example of Composition Association




```
class Title {
        …
}

class Window {
        private $title;
        function __construct() {
                $this->title = new Title();
                $this->title->setText(…);
        }
}
```

Window (i.e., the one with the filled diamond) contains a collection of Title, which Window "has more control over" (i.e., it instantiates Title)

Dependency and Association

- *Relationship* ::= *Dependency* || *Association*

- *Dependency* usually arises from
  - a local variable or
  - parameter variable
  - It's a weak and temporary relationship

- *Association* usually arise from
  - instance/member variables (attributes)

Extra Detail (not on the CS471 exam)

- *Association* ::= *UnspecifiedAssociation* ||
  Aggregation ||
  Composition

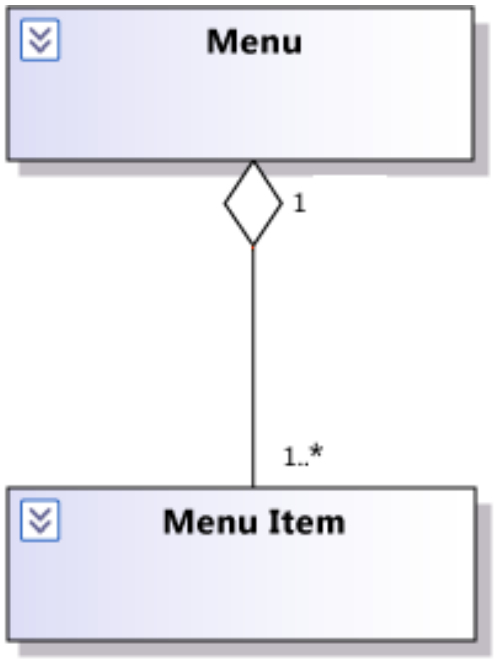
- *UnspecifiedAssociation* refers to an unspecified property

Extra Detail (not on the CS471 exam)

- *Aggregation* ::= refers to a weak *has-a* relationship.
  - ex: A **Section** has-a **Student**. **Student** objects are not destroyed when a **Section** is destroyed.

Extra Detail (not on the CS471 exam)

- *Aggregation* ::= refers to a weak *has-a* relationship.
  - ex: A **Section** has-a **Student**. **Student** objects are not destroyed when a **Section** is destroyed.
  - ex: A **Menu** has-a **Menu Item**. **Menu Item** objects are not destroyed when a **Menu** is destroyed.



Extra Detail (not on the CS471 exam)

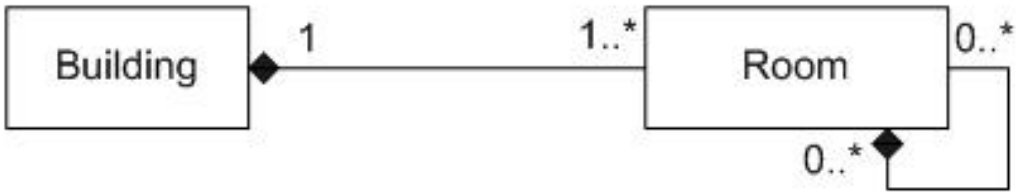
- *Aggregation* ::= refers to a weak *has-a* relationship.
  - ex: A **Section** has-a **Student**. **Student** objects are not destroyed when a **Section** is destroyed.
  - ex: A **Menu** has-a **Menu Item**. **Menu Item** objects are not destroyed when a **Menu** is destroyed.

- *Composition* ::= refers to a strong *has-a* relationship.

Extra Detail (not on the CS471 exam)

- *Aggregation* ::= refers to a weak *has-a* relationship.
  - ex: A **Section** has-a **Student**. **Student** objects are not destroyed when a **Section** is destroyed.
  - ex: A **Menu** has-a **Menu Item**. **Menu Item** objects are not destroyed when a **Menu** is destroyed.

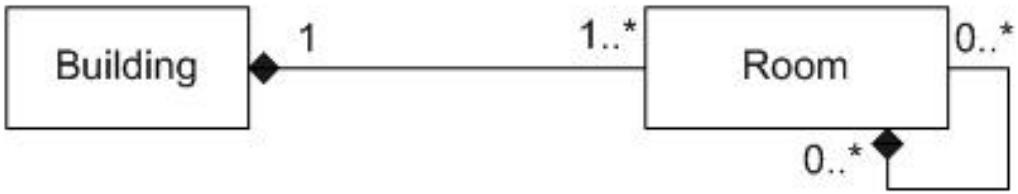
- *Composition* ::= refers to a strong *has-a* relationship.
  - ex: a **Building** has **Rooms**. If the **Building** is destroyed all **Rooms** are destroyed



Extra Detail (not on the CS471 exam)

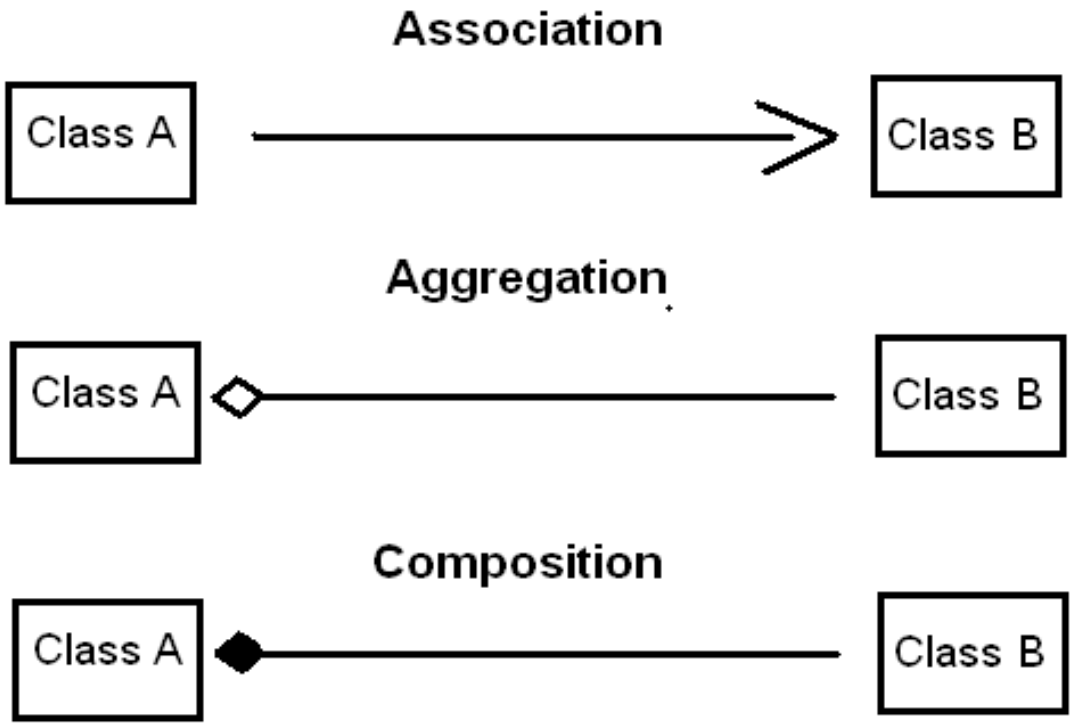
- *Aggregation* ::= refers to a weak *has-a* relationship.
  - ex: A **Section** has-a **Student**. **Student** objects are not destroyed when a **Section** is destroyed.
  - ex: A **Menu** has-a **Menu Item**. **Menu Item** objects are not destroyed when a **Menu** is destroyed.

- *Composition* ::= refers to a strong *has-a* relationship.
  - ex: a **Building** has **Rooms**. If the **Building** is destroyed all **Rooms** are destroyed

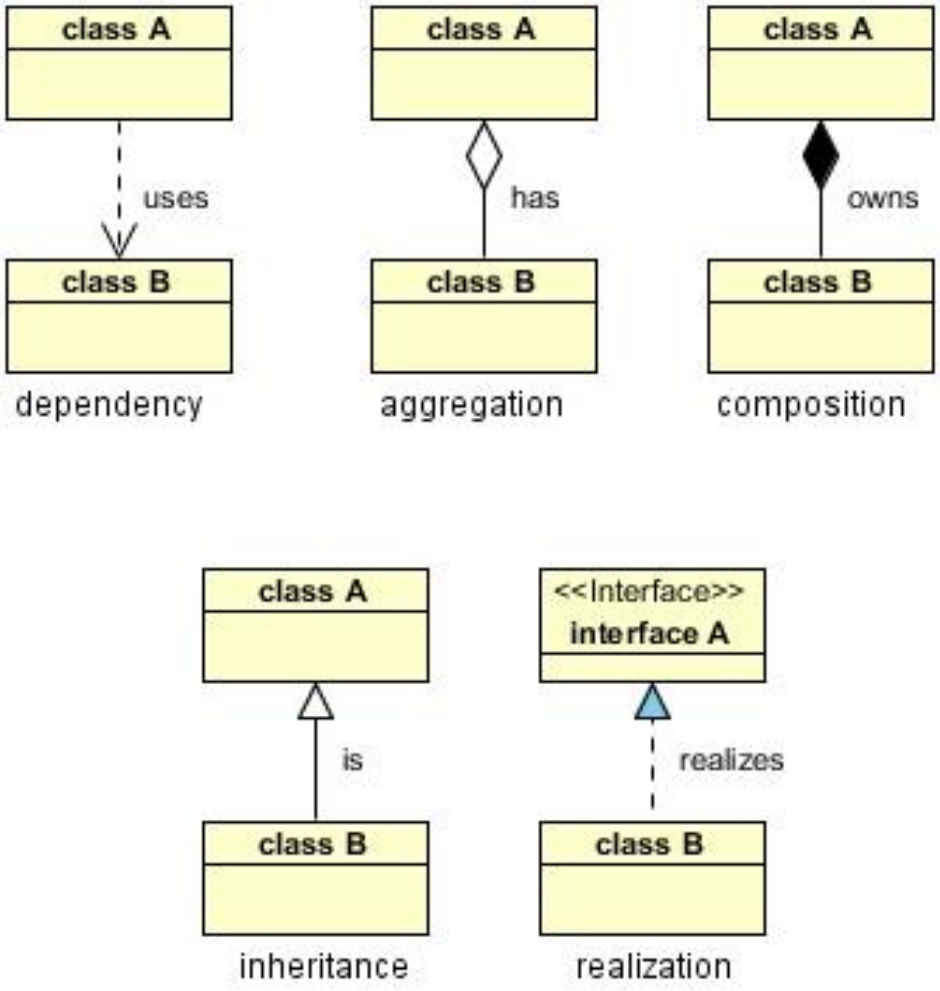


  - ex: If there were a relationship between **Course** and **Section** in the example, then all instances of a **Section** of a **Course** would be destroyed if the **Course** were destroyed.

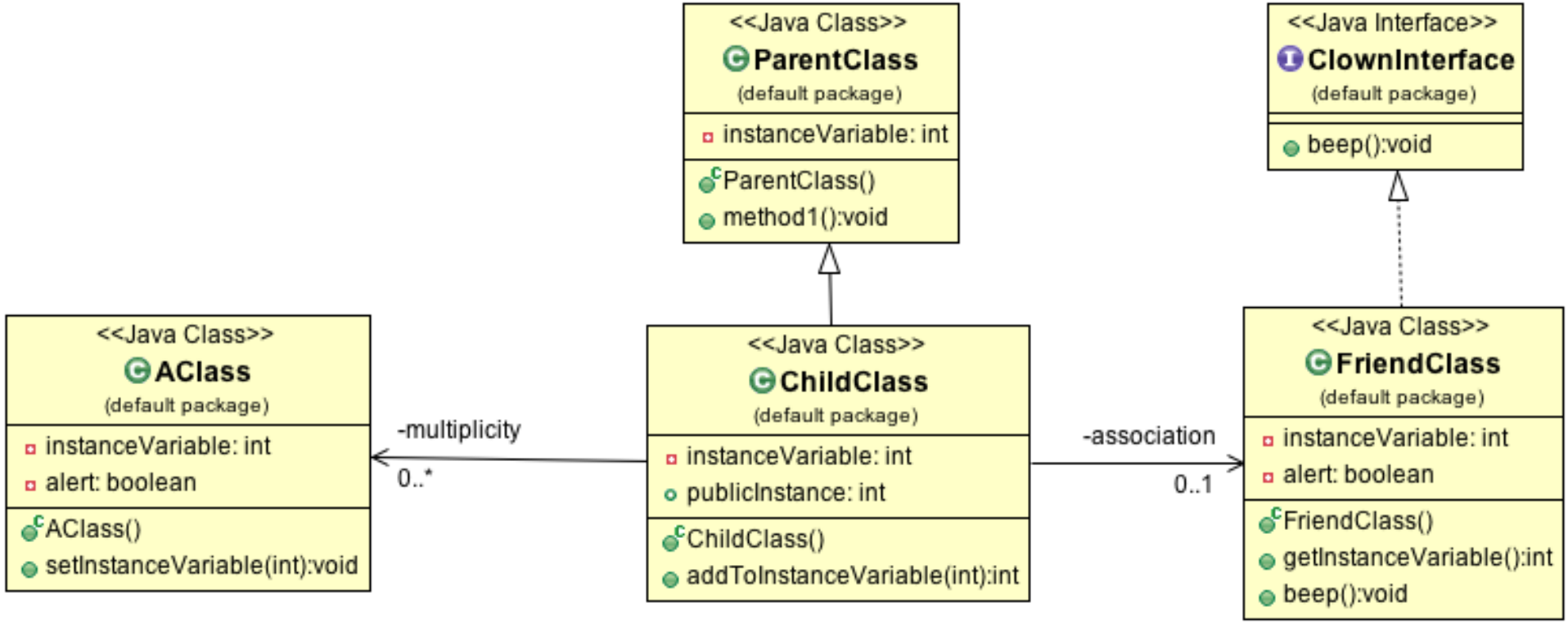
Association / Aggregation / Composition Notations



More Notations

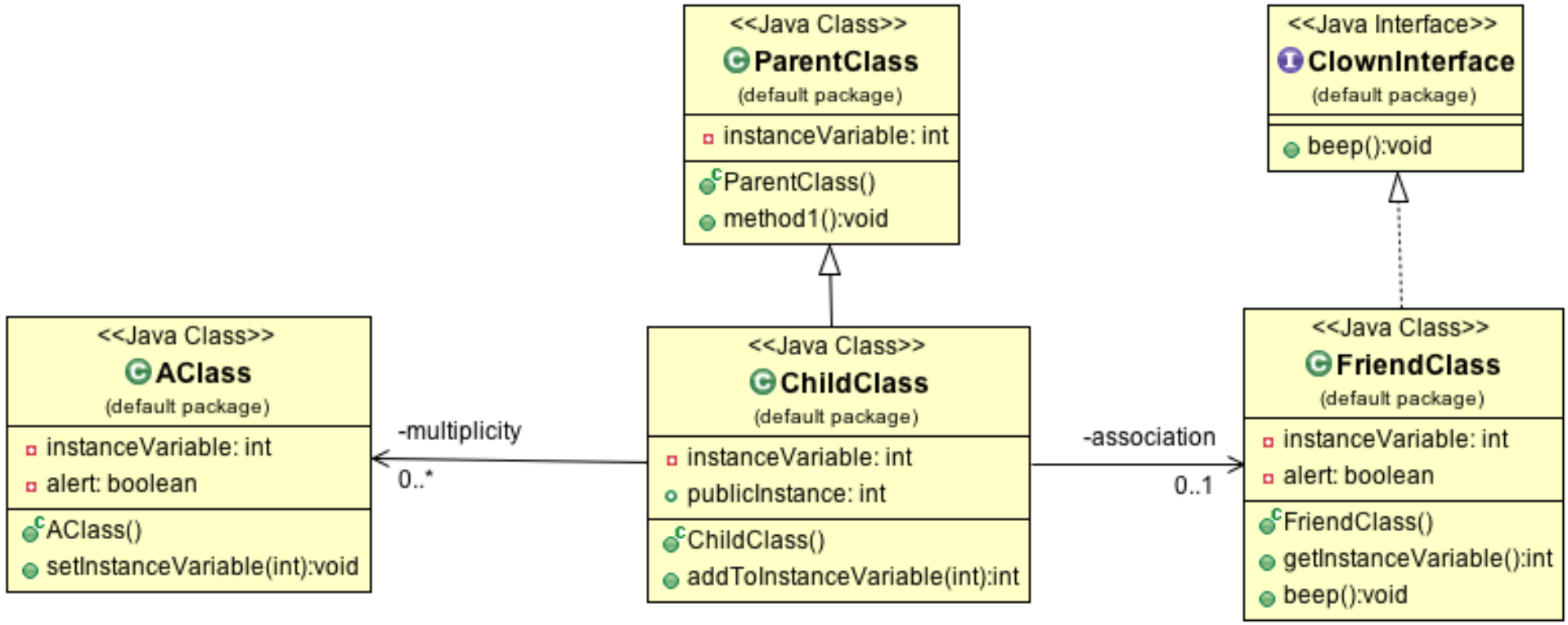


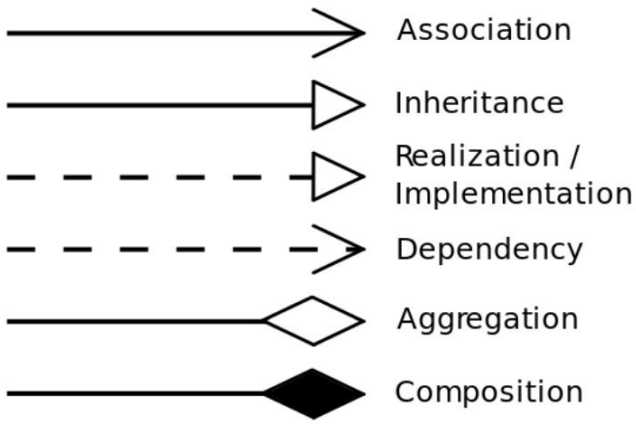
UML Class Diagram



What can you interpret?

UML Class Diagram





0..1	No instances, or one instance
1	Exactly one instance
0..*	Zero or more instances
*	Zero or more instances
1..*	One or more instances

UML Class Diagrams vs. Source Code

```
import java.util.Vector;

public class Driver {
    private StringContainer b = null;

    public static void main(String[] args){
        Driver d = new Driver();
        d.run();
    }

    public void run() {
        b = new StringContainer();
        b.add("One");
        b.add("Two");
        b.remove("One");
    }
}
```

UML Class Diagrams vs. Source Code

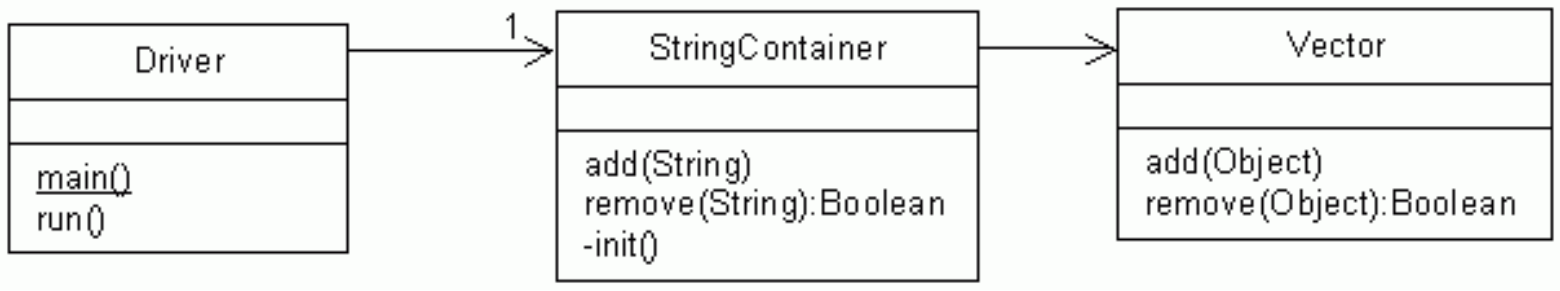
```
import java.util.Vector;

public class Driver {
    private StringContainer b = null;

    public static void main(String[] args){
        Driver d = new Driver();
        d.run();
    }

    public void run() {
        b = new StringContainer();
        b.add("One");
        b.add("Two");
        b.remove("One");
    }
}
```

```
class StringContainer {
    private Vector v = null;

    public void add(String s) {
        init();
        v.add(s);
    }

    public boolean remove(String s) {
        init();
        return v.remove(s);
    }

    private void init() {
        if (v == null)
            v = new Vector();
    }
}
```

UML Class Diagrams vs. Source Code

```
import java.util.Vector;

public class Driver {
    private StringContainer b = null;

    public static void main(String[] args){
        Driver d = new Driver();
        d.run();
    }

    public void run() {
        b = new StringContainer();
        b.add("One");
        b.add("Two");
        b.remove("One");
    }
}
```

```
class StringContainer {
    private Vector v = null;

    public void add(String s) {
        init();
        v.add(s);
    }

    public boolean remove(String s) {
        init();
        return v.remove(s);
    }

    private void init() {
        if (v == null)
            v = new Vector();
    }
}
```



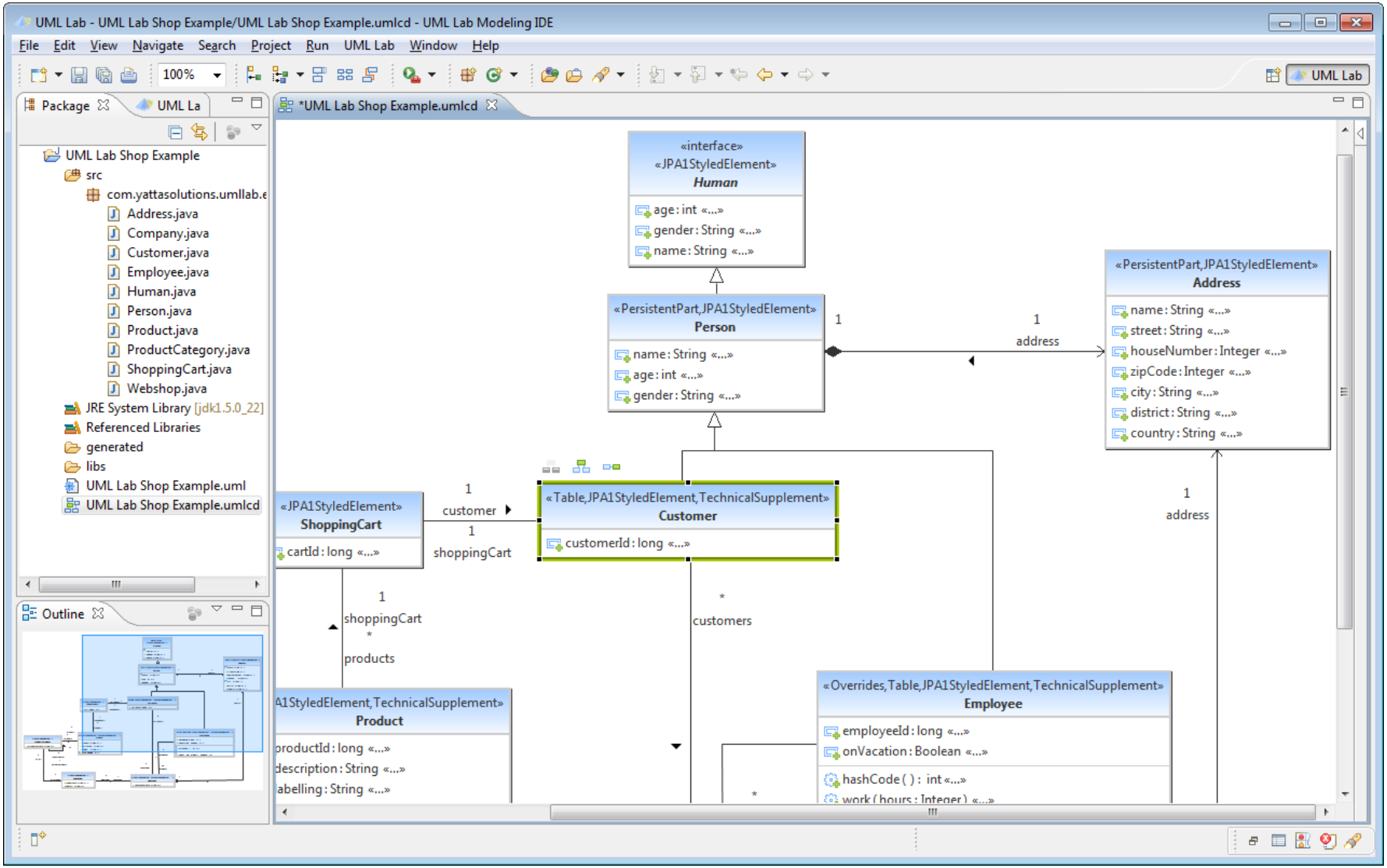
Generate UML Class Diagram from Source Code (Whiteboard only)

```
import java.util.ArrayList;

public class Zot {
    protected int effort;
}

public class Foo {
    private String name;
    private ArrayList<Zot> backlog;
}
```

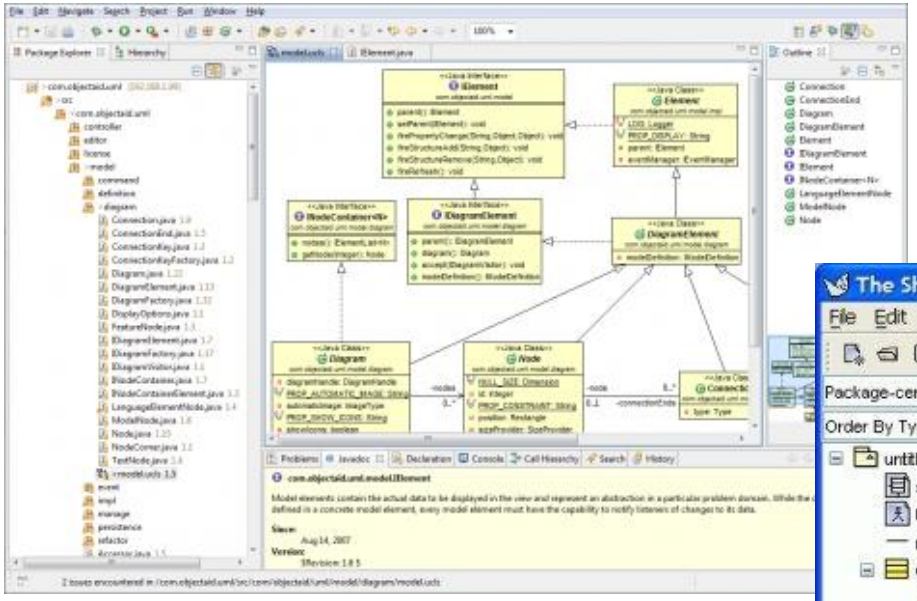
Documenting a Finished Implementation

- Diagrams completed after-the-fact
- Requirement for the diagrams may be contractual
- Purpose is to introduce a new engineer to the existing implementation
- Might be auto-generated by a UML tool
  - IBM Rational Rose
  - ArgoUML (http://argouml.tigris.org/)

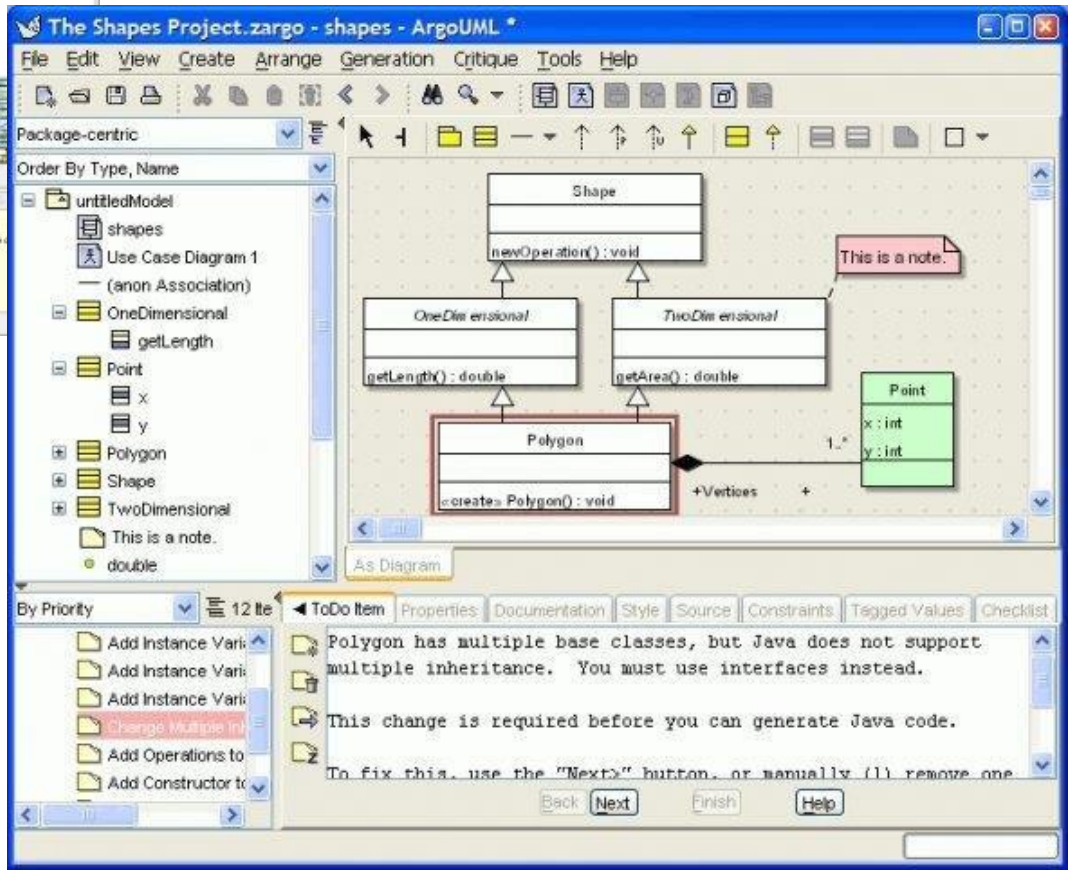
Documenting a Finished Implementation



Documenting a Finished Implementation

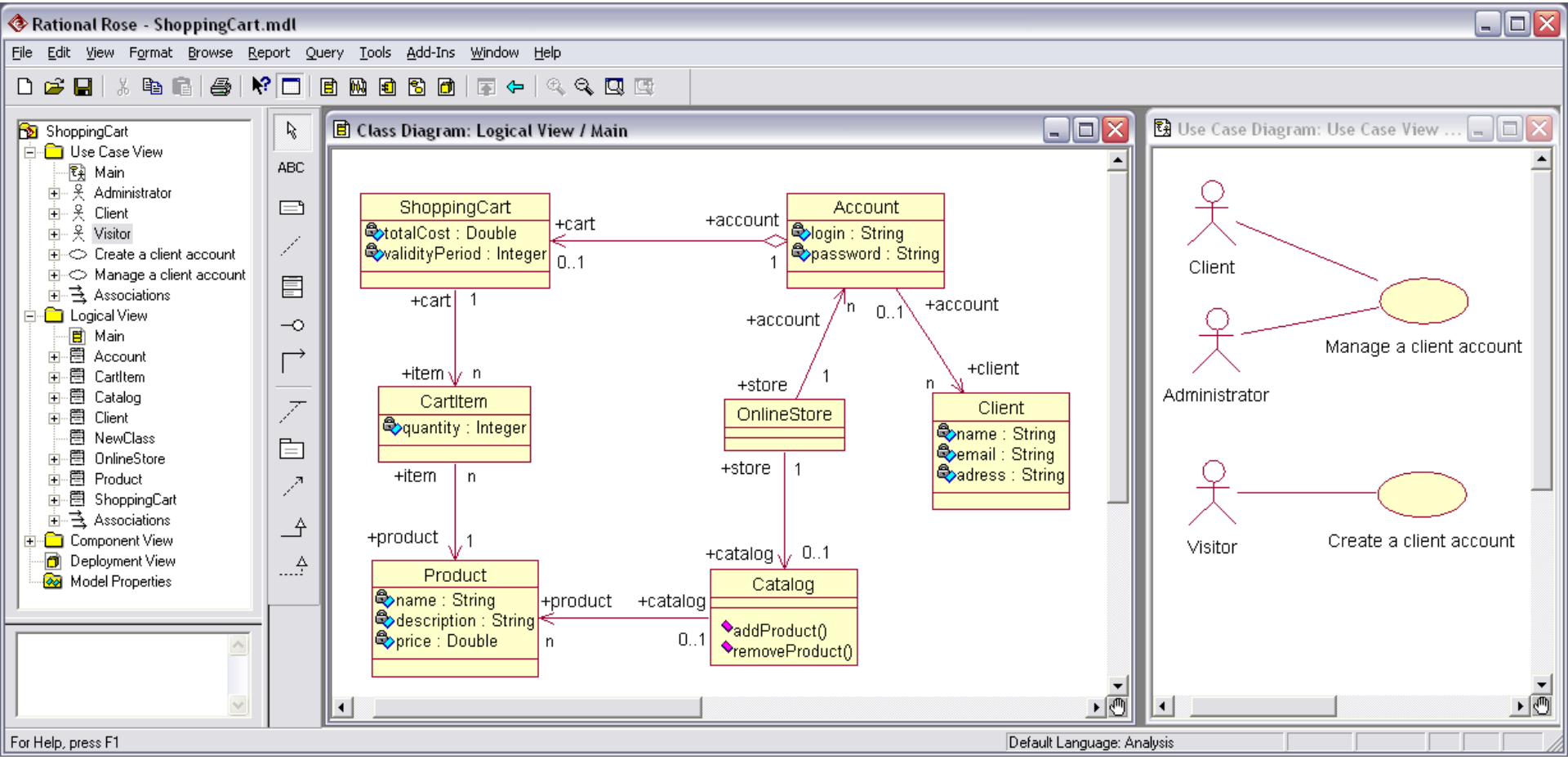


http://argouml.tigris.org/



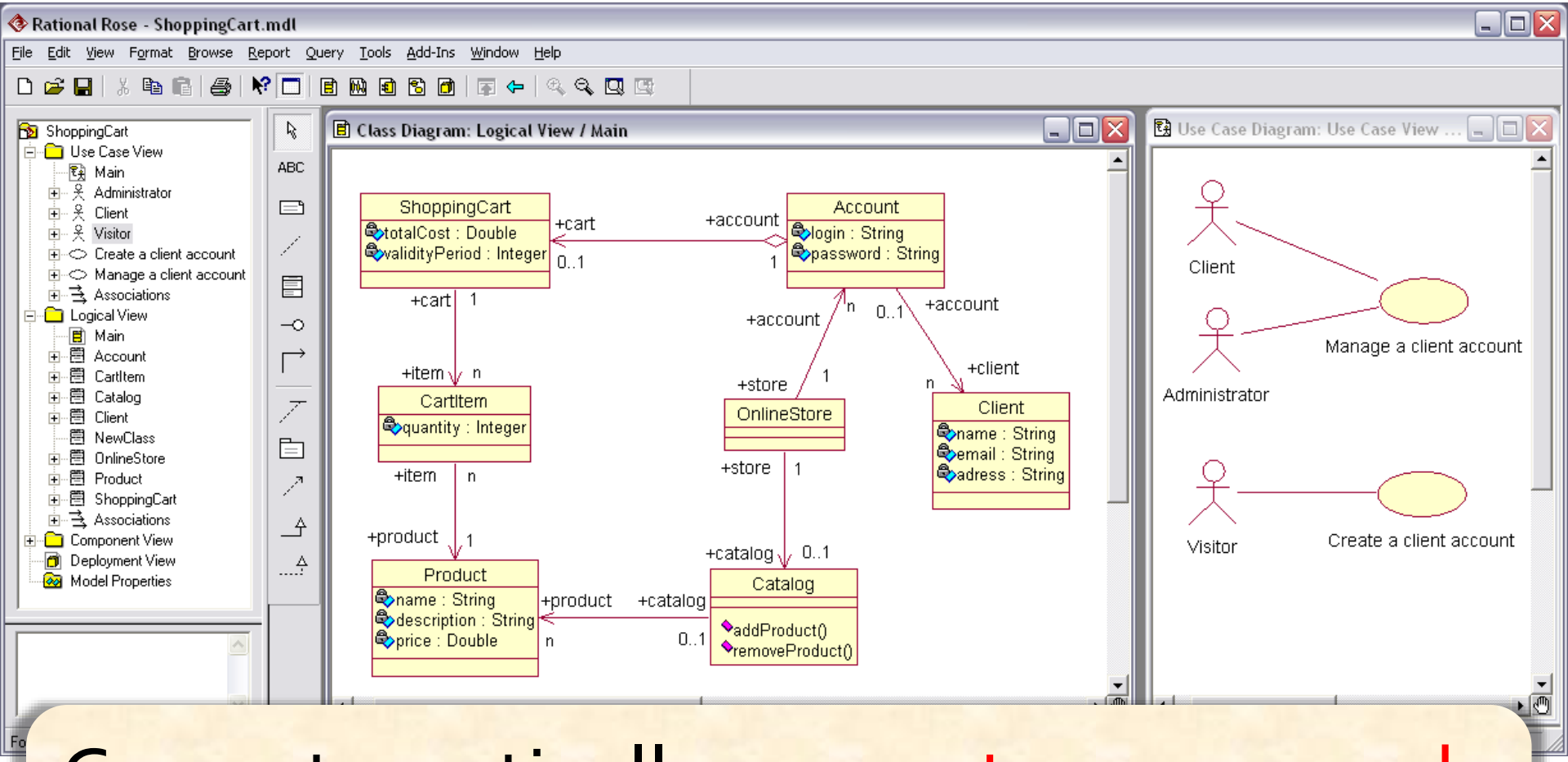
Documenting a Finished Implementation

IBM Rational Rose



Documenting a Finished Implementation

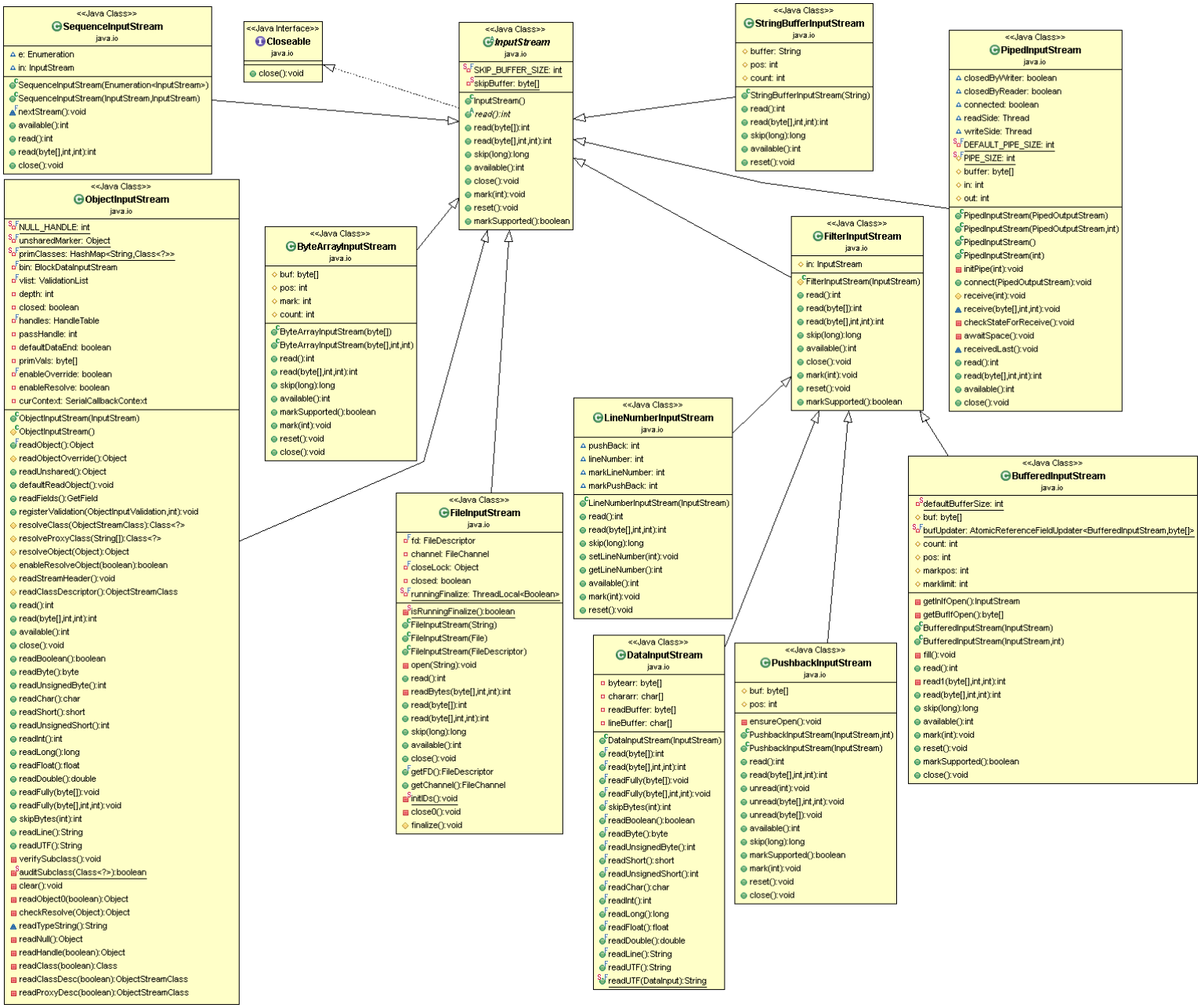
IBM Rational Rose



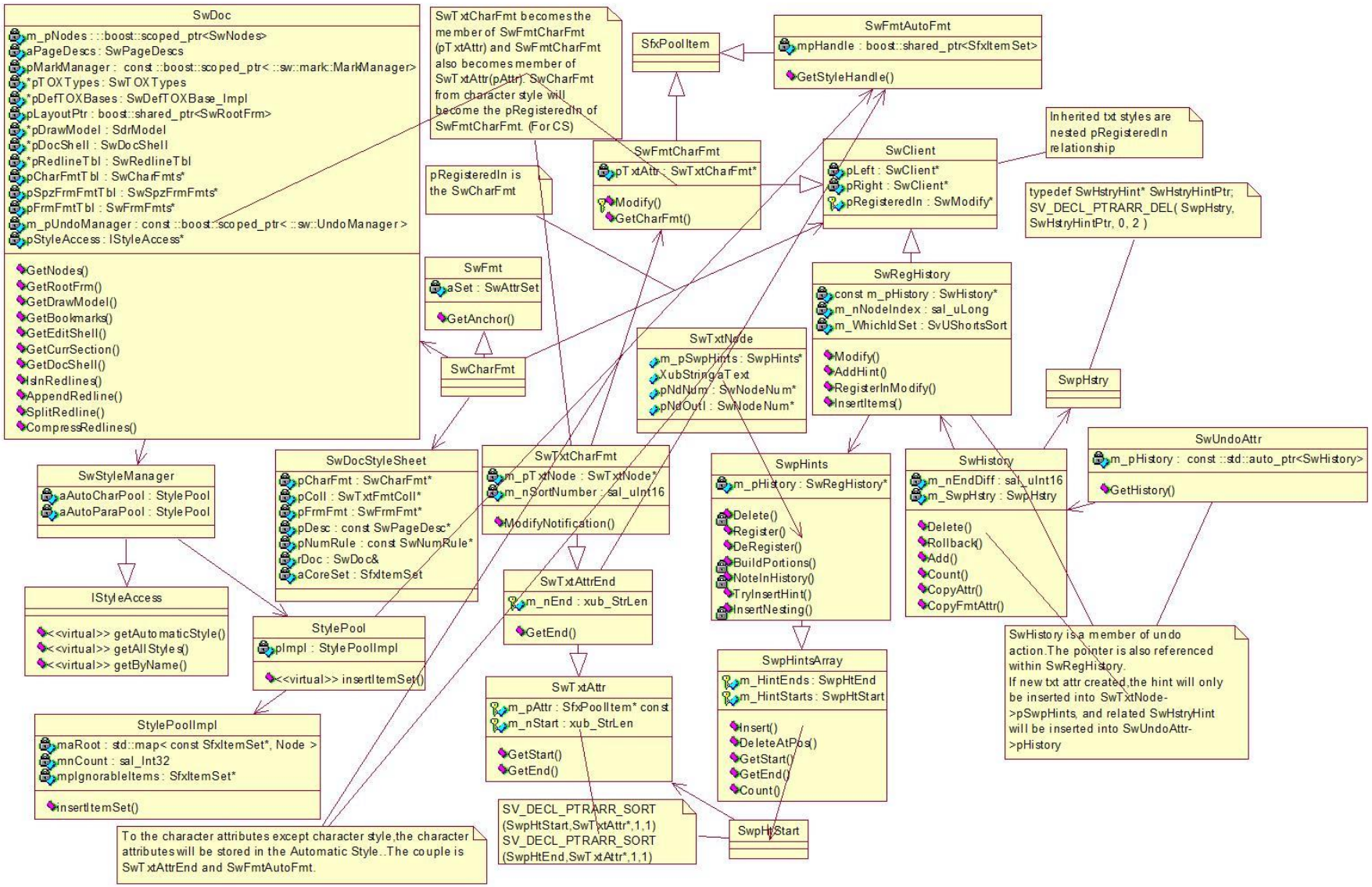
Can automatically generate source code based on the class diagrams

Limitations of UML Class Diagrams

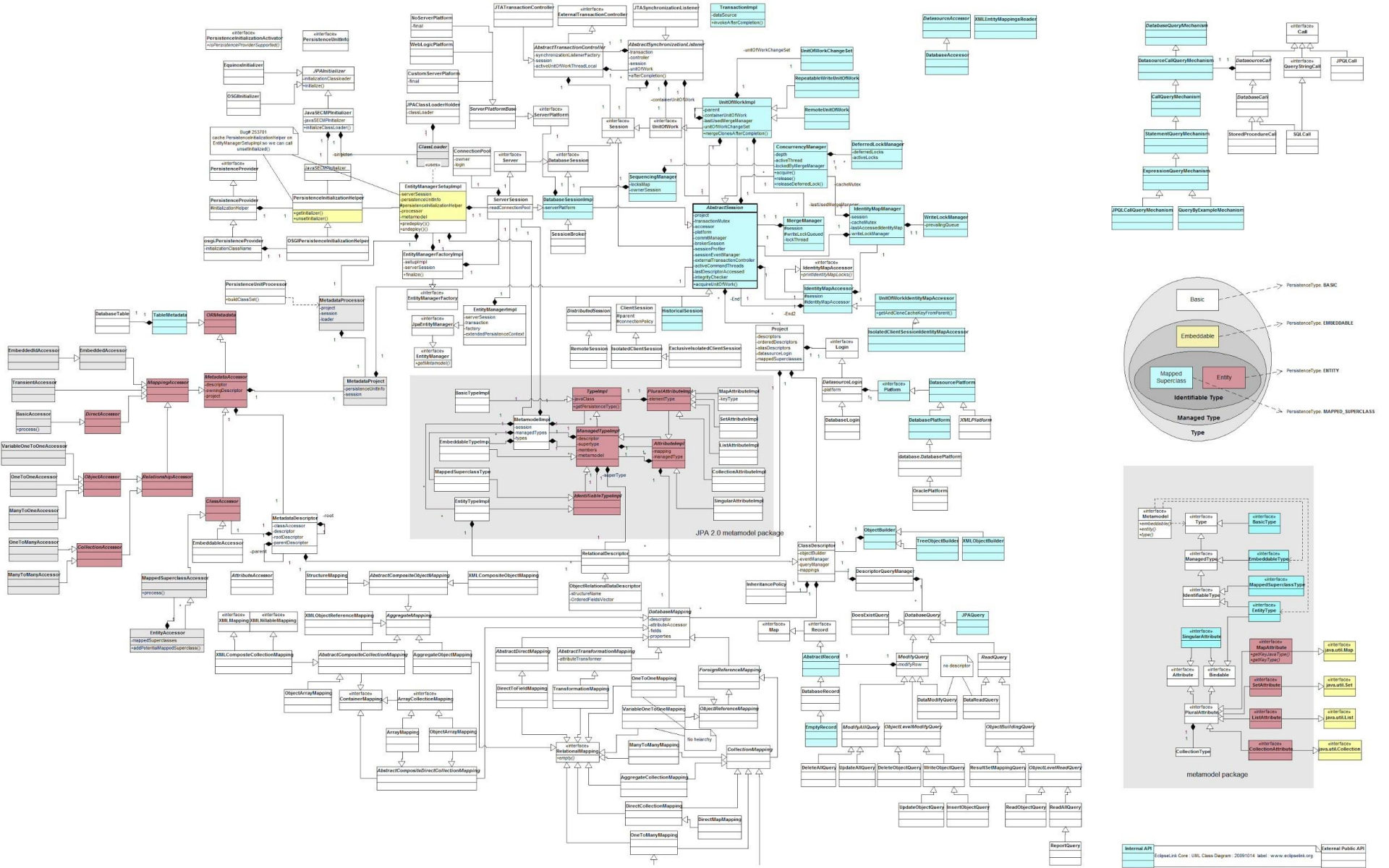
Limitations of UML Class Diagrams



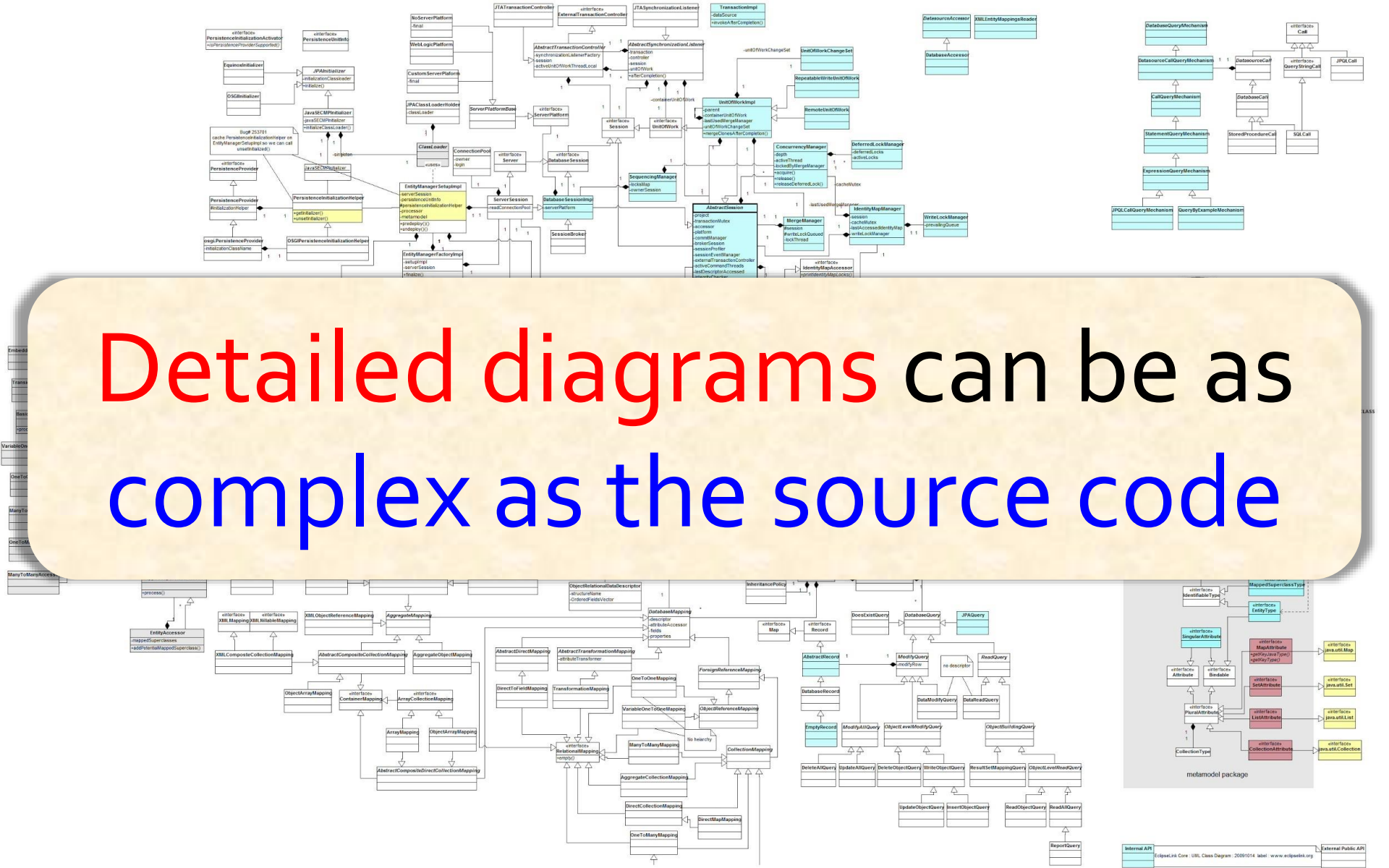
Limitations of UML Class Diagrams



Limitations of UML Class Diagrams



Limitations of UML Class Diagrams



Detailed diagrams can be as complex as the source code

In other words…

"The map is not the territory"*

- The diagram is not supposed to be the source code

- If the diagram is too complex and contains all the details from the source code, it is
  - not abstract enough
  - less useful

*Alfred Korzybski

UML in Industry

- Useful for documenting class relationships
  - without describing how their objects interact
  - nor how they work internally

- May be greatly simplified on a whiteboard
  - Interesting data
  - Interesting methods

- Objective is to communicate a particular agenda, not document everything

UML Class Diagrams: Key Points for CS471

- Always show all instance variables and public methods
- Show class associations (usually arise from instance/member variables)
- Show inheritance
- Interfaces
- Not required:
  - Method parameters (unless essential for understanding… like overloading)
  - dependencies (usually arising from local variables)

Class Diagrams Summary

- Graphical illustration of relationships between classes

- Static (doesn't provide information about how the system executes)

- Documents
  - a few details about the interface to a class
  - a bit of its internal structure (attributes & methods)