Department of
Computer Science

Software Design Review

Software Design

- Objectives:
  - Fulfill functionality
  - Achieve quality attributes
- Reflect designer's thoughts in code. Good design requires:
  - **Follow good design principles**
  - Solid programming skills
- Software Design in General
  - Data structure, interfaces and logic
- Software Design in OO
  - Classes with data structure, interfaces and logic

The fundamentals

- Abstraction and Modularization
  - How to apply Abstraction and Modularization in
    - Software Design
    - OO Design
- Design principles
  - General design principles:
    - Information Hiding—Hide data structure, hide details, hide variations
    - Low coupling, High cohesion
    - Least of Knowledge
  - OO design principles (SOLID)
    - Single Responsibility Principle
    - Open-Closed Principle
    - Liskov Substitution Principle
    - Interface Segregation Principle
    - Dependency Inversion Principle

General Design Principles

- Information Hiding
  - Hide data
    - Do not expose internal data structure!
    - Be careful about getter and setter!
  - Hide details
  - Hide variation
- High cohesion, low coupling
  - Methods should only access data within their class
  - A function should always access the same data.
- Least of Knowledge

SOLID

- Single Responsibility Principle: there can be only one reason for a class to change.
  - Use separate classes for different set of functionality
  - No function should have two purposes.
- Open-Closed Principle: Classes and methods should be open for extension but closed for modification.
  - Always think about future changes
- Liskov Substitution Principle: Every function or method which expects an object parameter of class A must be able to accept a subclass of A as well, without knowing it.
  - Forget Is-a when you design an heritance hierarchy
- Interface Segregation Principle: Classes should not depend on interfaces that they not use.
  - Interface design should also have high cohesion
  - Single responsibility principle applied at Interface design
- Dependency Inversion Principle: *High level classes should not depend on low level classes. Both should depend upon abstractions. Details should depend upon abstractions. Abstractions should not depend upon details.*
  - Favor composition over inheritance
  - Try you best not to inherit from a common class
  - Use interfaces to abstract the commonality between classes, and only depend on the interfaces

Single Responsibility Principle

- There can be only one reason for a class to change.
  - Use separate classes for different set of functionality
  - No function should have two purposes.

Open-Closed Principle

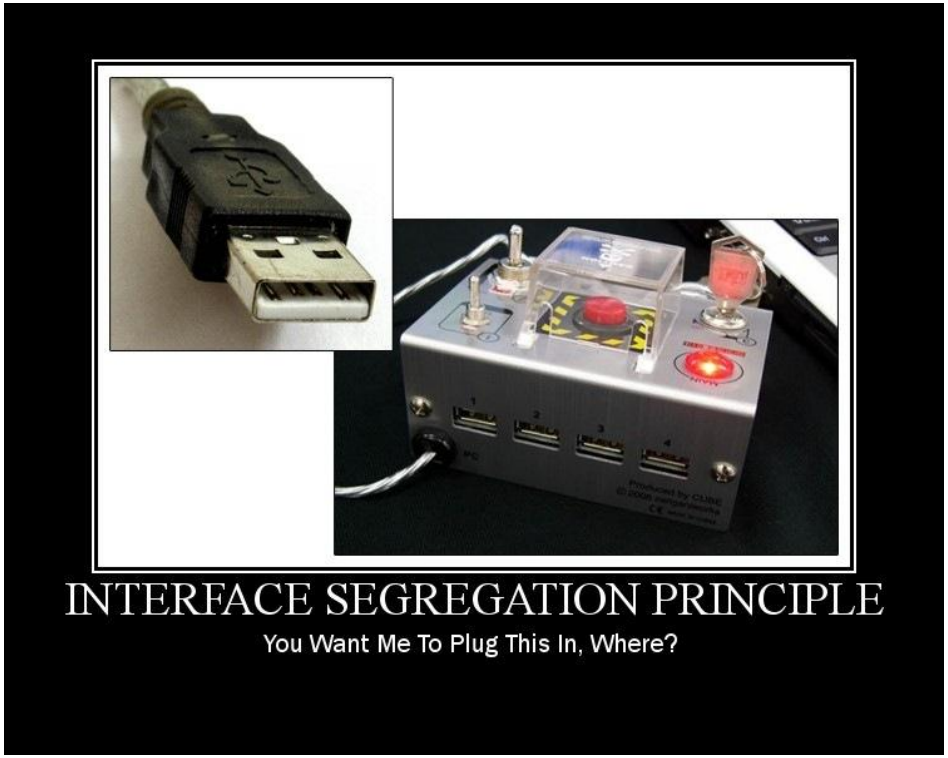
- Classes and methods should be open for extension but closed for modification.
  - Always think about future changes

Liskov Substitution Principle

- Every function or method which expects an object parameter of class A must be able to accept a subclass of A as well, without knowing it.
  - Forget Is-a when you design an heritance hierarchy

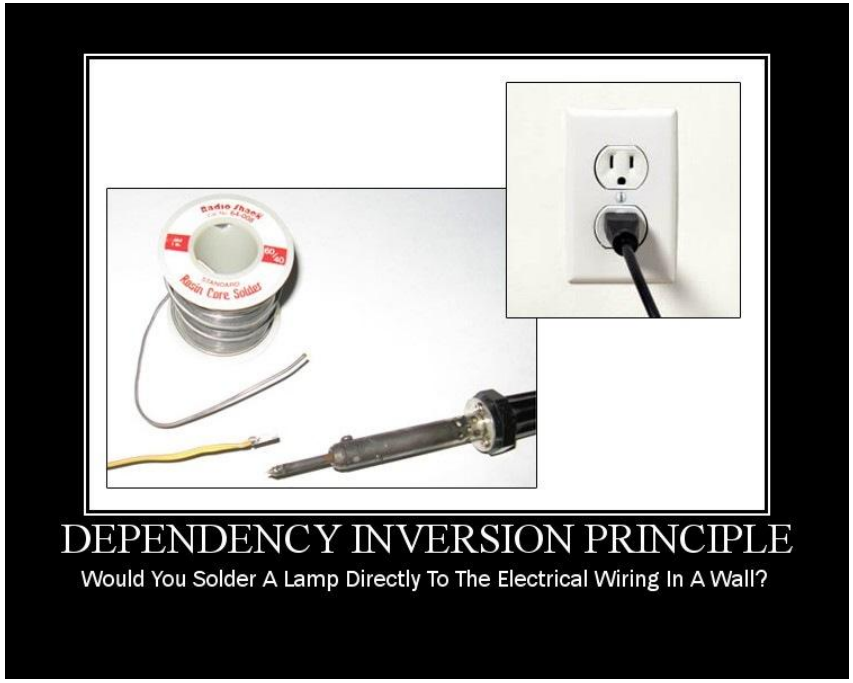
Interface Segregation Principle

- Classes should not depend on interfaces that they not use.
  - Interface design should also have high cohesion
  - Single responsibility principle applied at Interface design



Dependency Inversion Principle

- *High level classes should not depend on low level classes. Both should depend upon abstractions. Details should depend upon abstractions. Abstractions should not depend upon details.*
  - Favor composition over inheritance
  - Try not to inherit from a common class
  - Use interfaces to abstract the commonality between classes, and only depend on the interfaces



Localize and minimize changes

- A design can only follow these principles under ***some*** circumstances
- Changes can only be localized to certain extent
- Crosscutting changes are inevitable

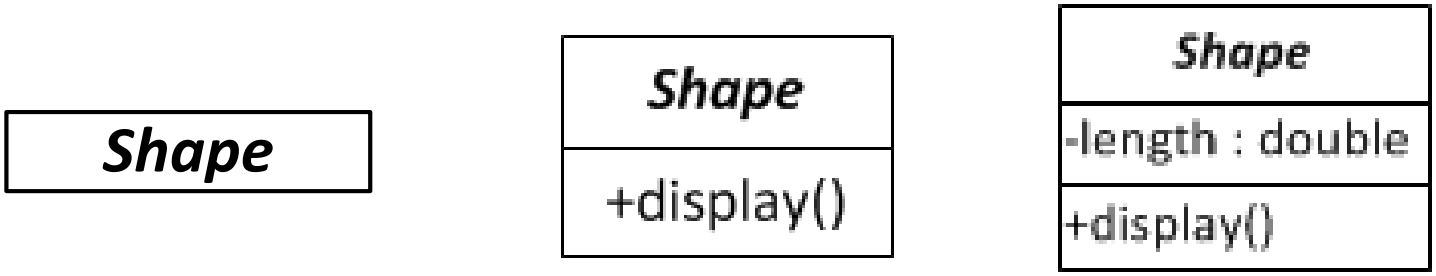
The Unified Modeling Language (UML)

- A standard language for
  - specifying, visualizing, constructing, and documenting the artifacts of software systems,
  - business modeling and other non-software systems.
- The UML represents
  - a collection of best engineering practices
  - that have proven successful in the modeling of large and complex systems.[1]
- The UML helps project teams
  - communicate, explore potential designs,
  - validate the architectural design of the software.

The UML Diagrams

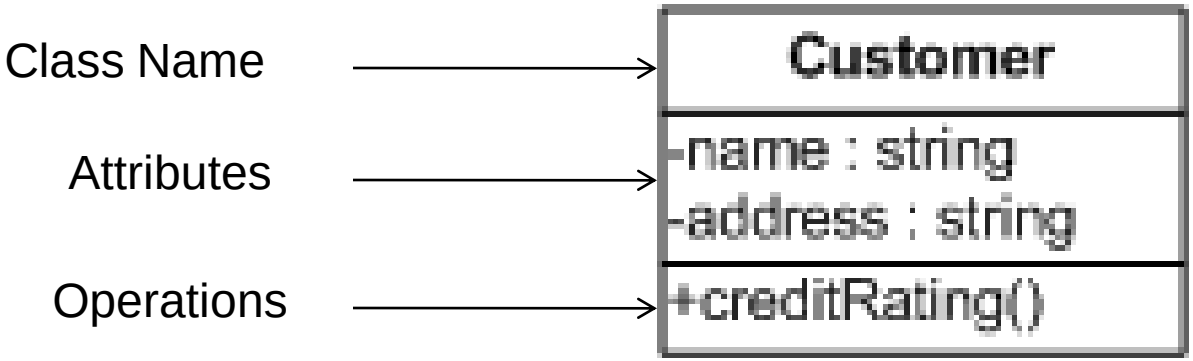
- Use Case Diagram
- Class Diagram
- Interaction Diagrams
  - Sequence Diagram
  - Collaboration Diagram
- State Diagram
- Activity Diagram
- Physical Diagrams
  - Component Diagram
  - Deployment Diagram

Class Diagram

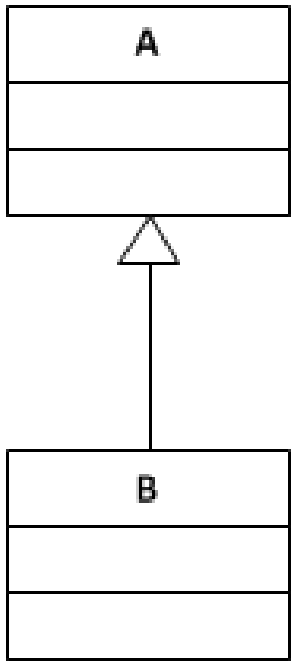


The Class Diagram –A Class

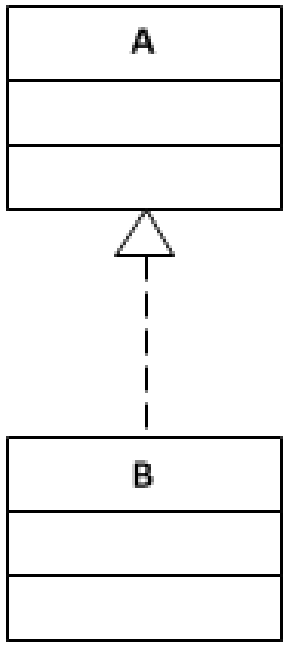
models class structure and contents using design elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others.



UML Class Relationships

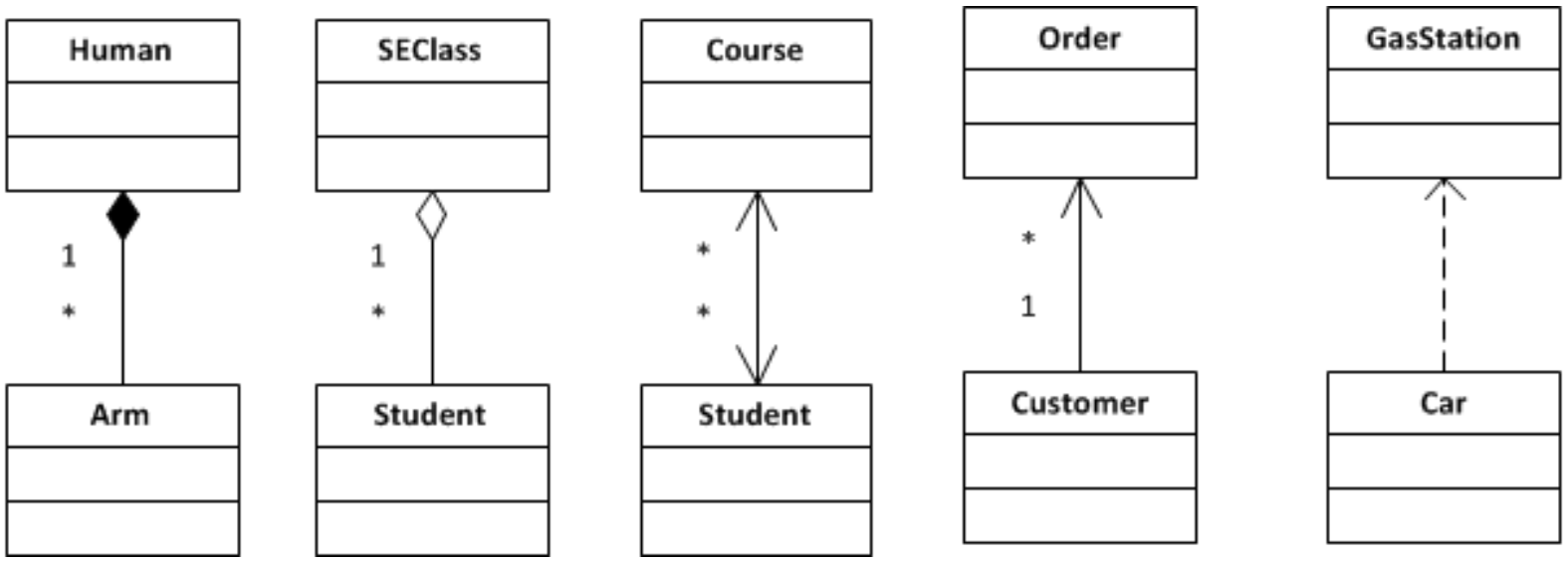


B is derived from A
A generalizes B



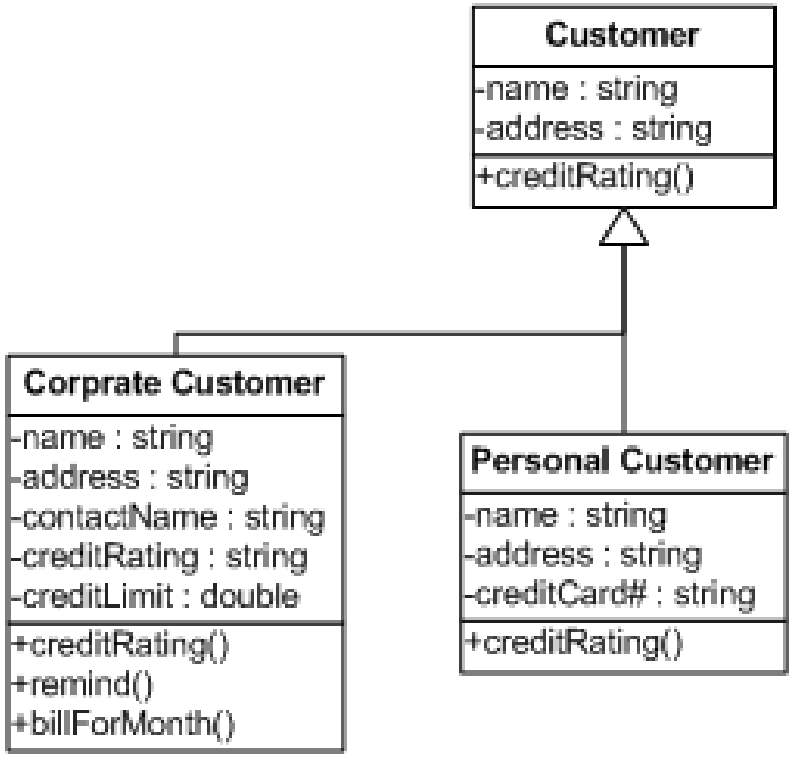
B realizes the interfaces defined in A

UML Class Relationships

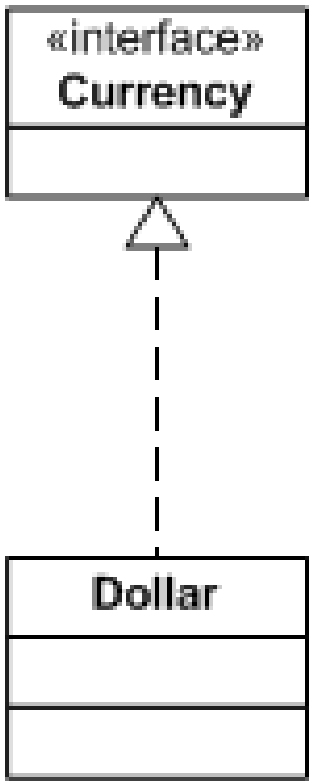


Composition Aggregation Bi-Association Uni-Association Dependency

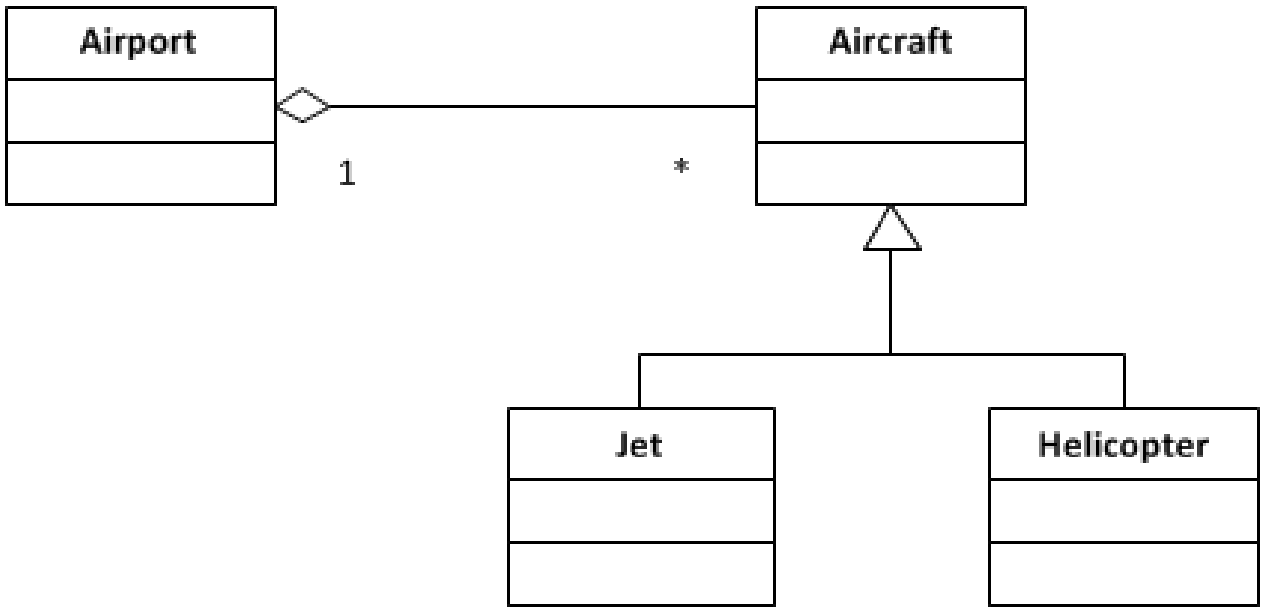
The Class Diagram – Implementation Inheritance (Generalize/Specialize)



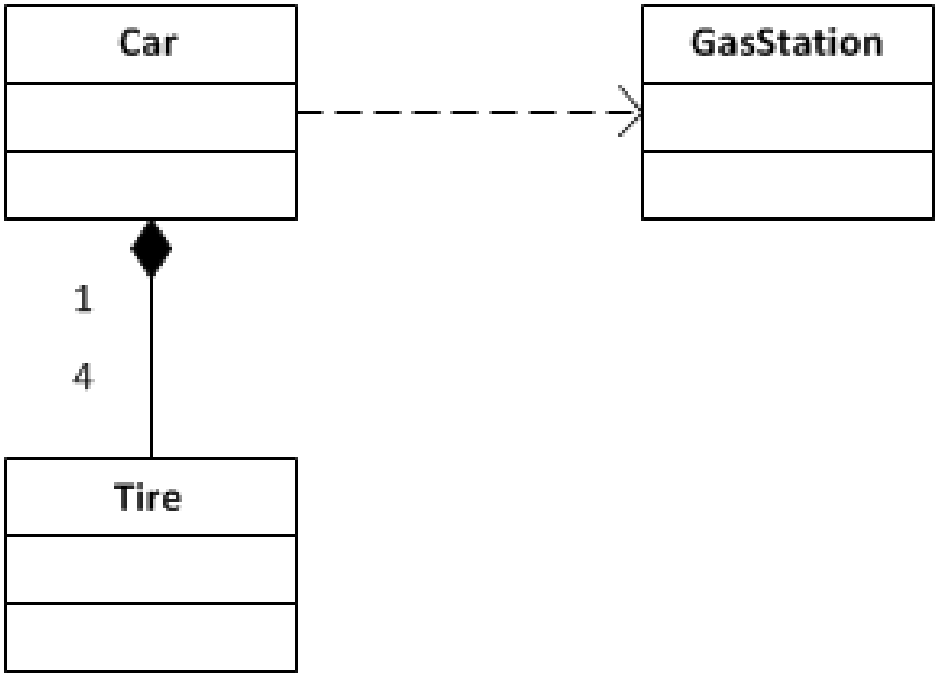
The Class Diagram – Interface Inheritance (Specifies/Redefines/Realizes)



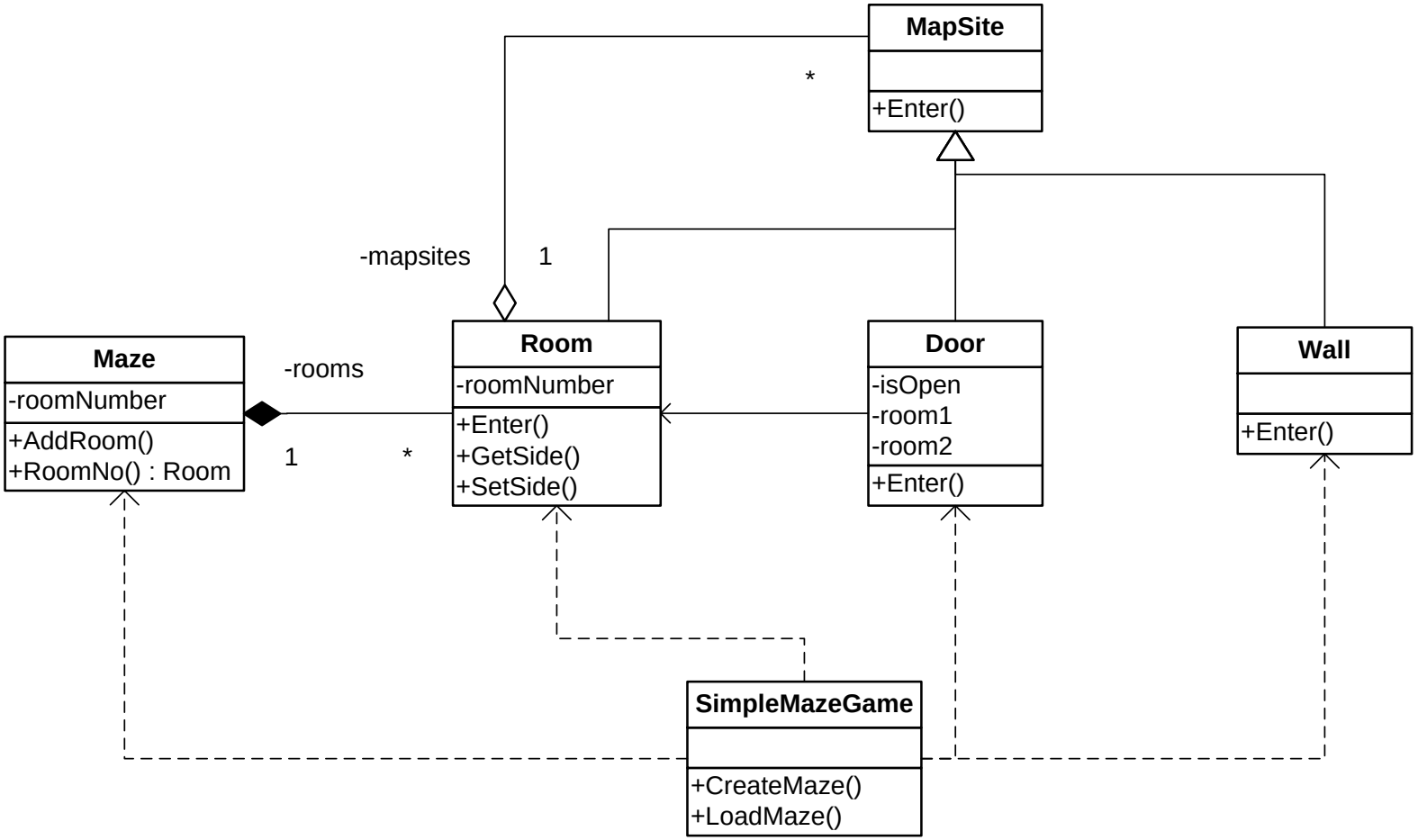
UML Class Diagram



UML Class Diagram



The Maze Game UML



UML Tools

- Proprietary
  - Rational Rose
  - Visio
  - OmniGraffle
- Open Source
  - **Dia (http://live.gnome.org/Dia)**
  - Eclipse MDT UML2 (http://www.eclipse.org/modeling/mdt/downloads/?project=uml2)
  - BOUML (http://bouml.free.fr)
  - ArgoUML (http://argouml.tigris.org)

References

- UML Distilled by Martin Fowler

- http://www.holub.com/goodies/uml