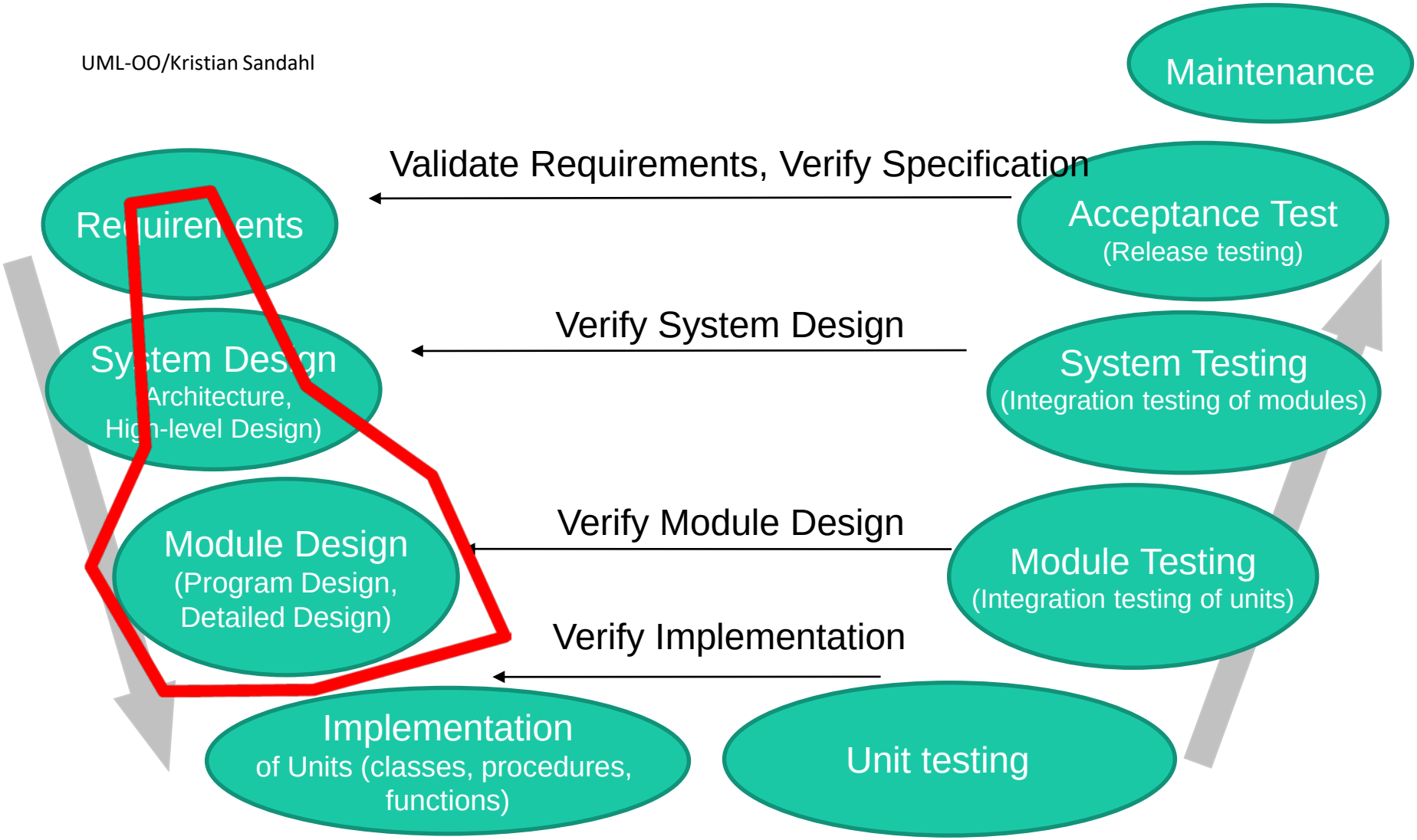
UML and Object-orientation

Kristian Sandahl

LINKÖPING UNIVERSITY

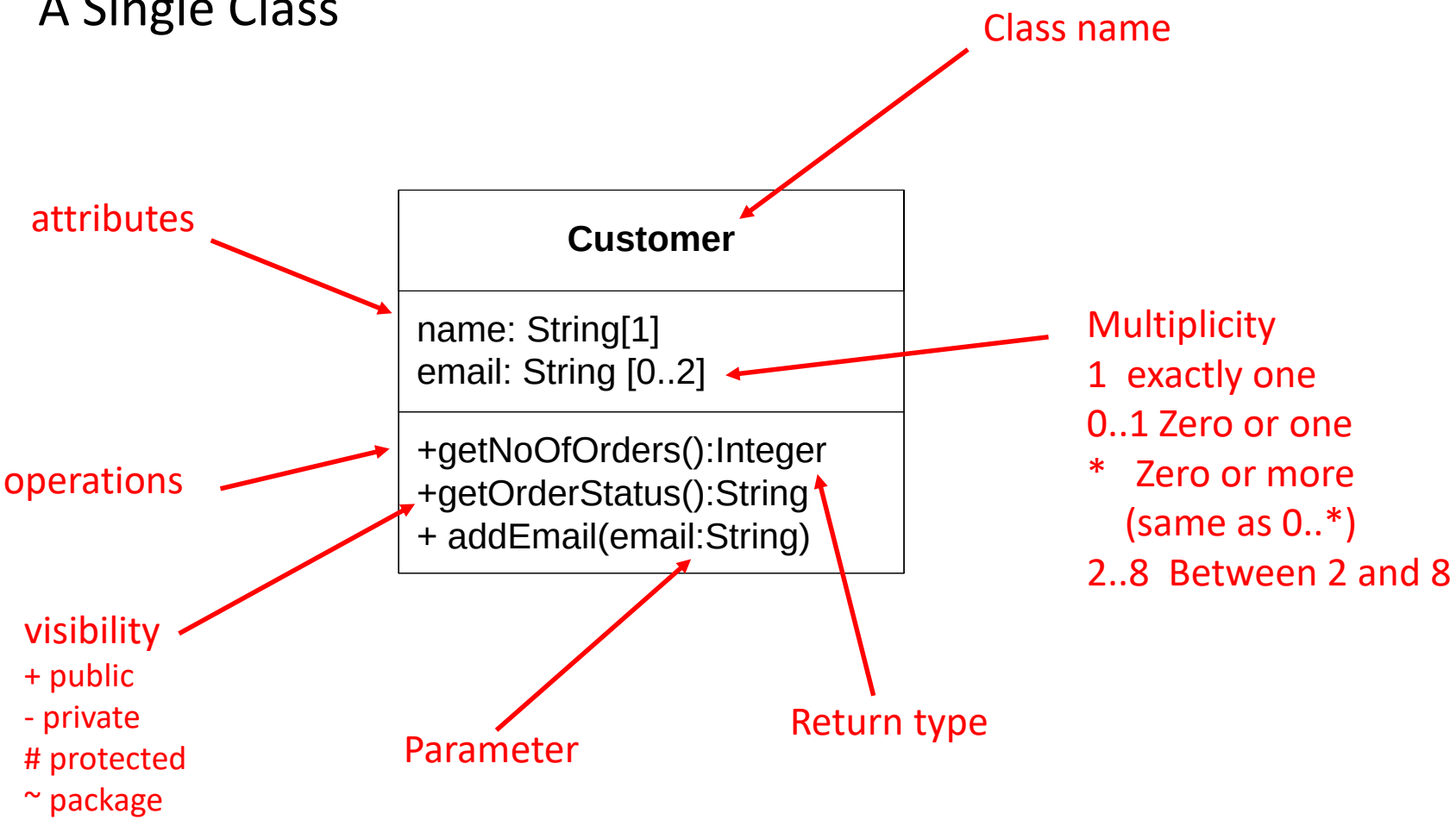


Project Management, Software Quality Assurance (SQA), Supporting Tools, Education

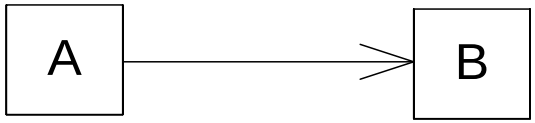
The goals of module design

- Provide the expected function
- Prepare for change:
  - Separation of concern
  - Testability
  - Understandability
- Contribute to quality, eg:
  - Performance
  - Usability
  - Reliability
  - ...
- Map for the implementers and testers

A Single Class

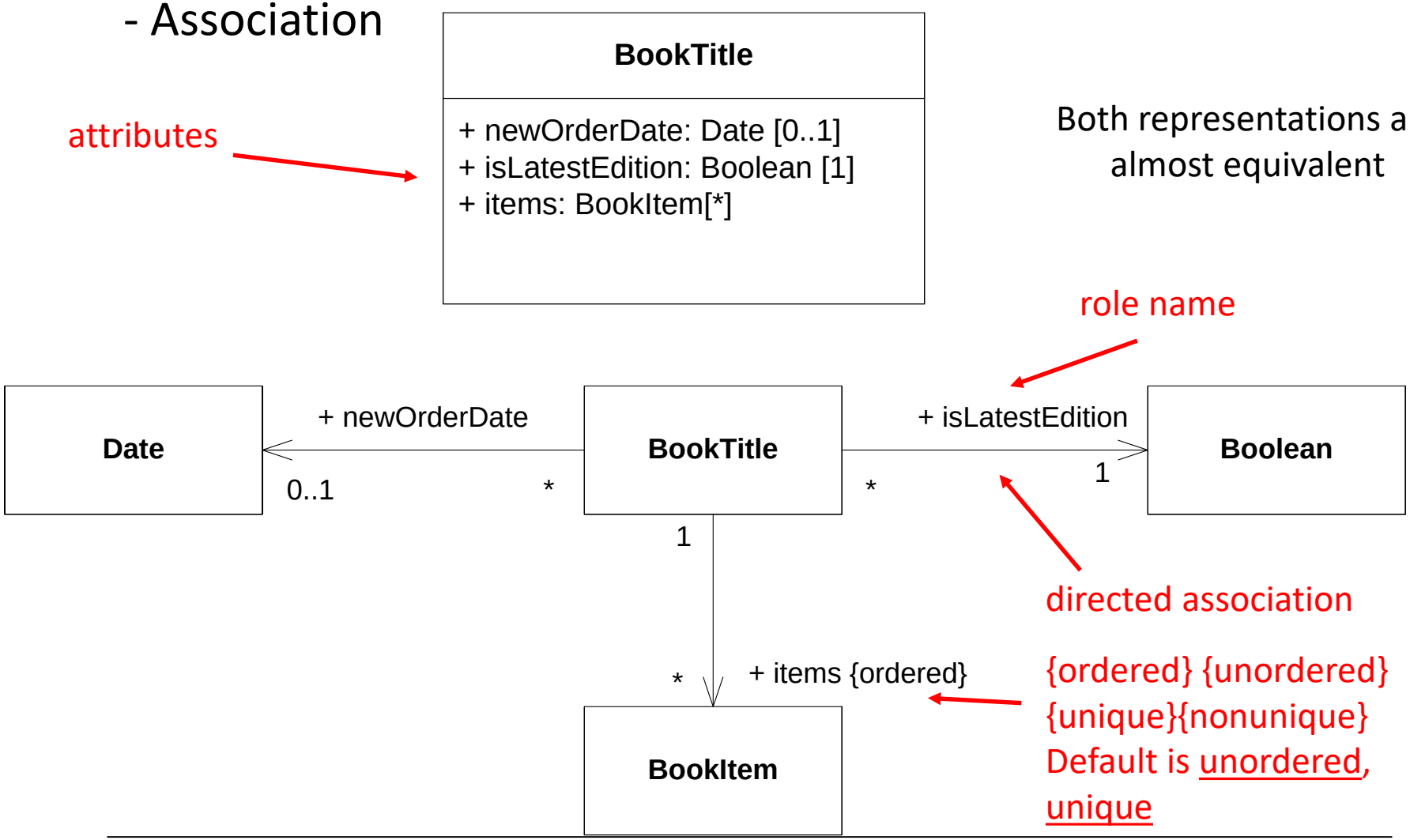


Relationships (1/6) - overview and intuition - Association

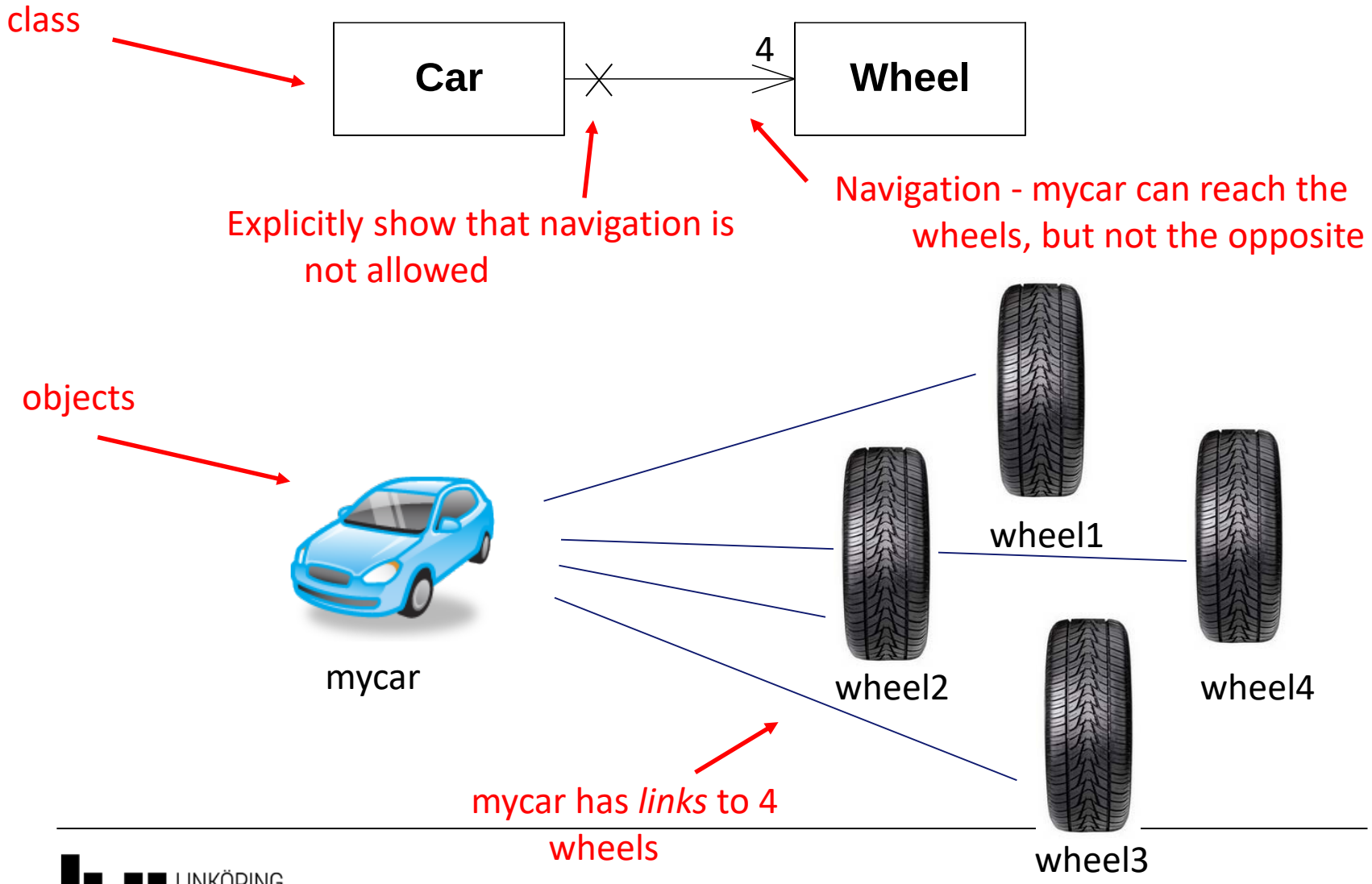


Association
(with navigability)

Relationships (1/6) - overview and intuition - Association



Relationships (1/6) - overview and intuition - Association



Relationships (1/6) - overview and intuition - Association

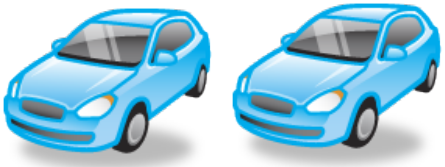
What does it mean to have a * here?

What if we have multiplicity 1 instead?

"*"

A wheel can be linked to more than one car instance

"1"



A wheel can only be liked to one car instance

Relationships (1/6) - overview and intuition - Association

Associations are the "glue" that ties a system together

mycar1

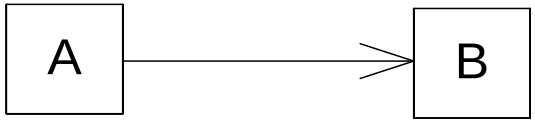
association instance = *link*

An association describes a *relation* between objects at run-time.

wheel1 wheel2 wheel3 wheel4

{(mycar1,wheel1),
(mycar1,wheel2),
(mycar1,wheel3),
(mycar1,wheel4)}

Relationships (2/6) - overview and intuition - Aggregation



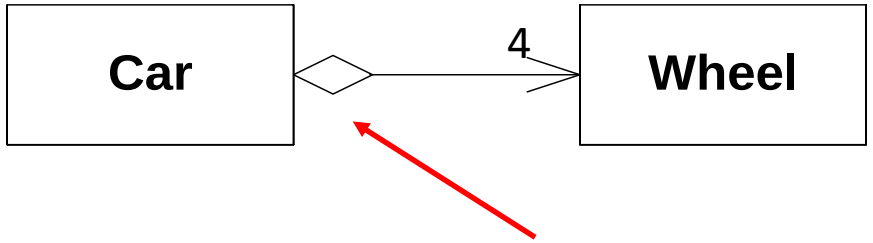
Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

Aggregation

Relationships (2/6) - overview and intuition - Aggregation

Common vague interpretations: "owns a" or "part of"



What does this mean? What is the difference to association?

Vague definitions → Inconsistency and misunderstandings

Aggregation was added to UML with little semantics. Why?

Jim Rumbaugh
"Think of it as a modeling placebo"

Recommendation: - Do not use it in your models.
- If you see it in other's models, ask them what they actually mean.

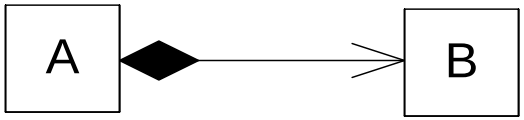
Relationships (3/6) - overview and intuition - Composition

Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

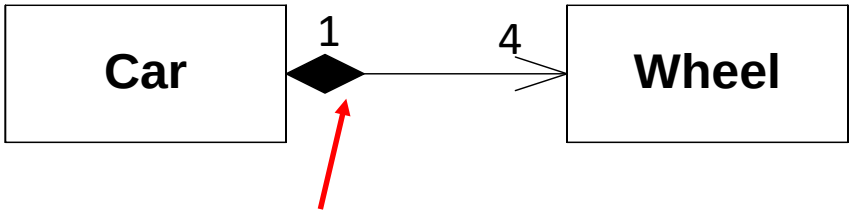
Aggregation

Avoid it to avoid misunderstandings



Composition

Relationships (3/6) - overview and intuition - Composition

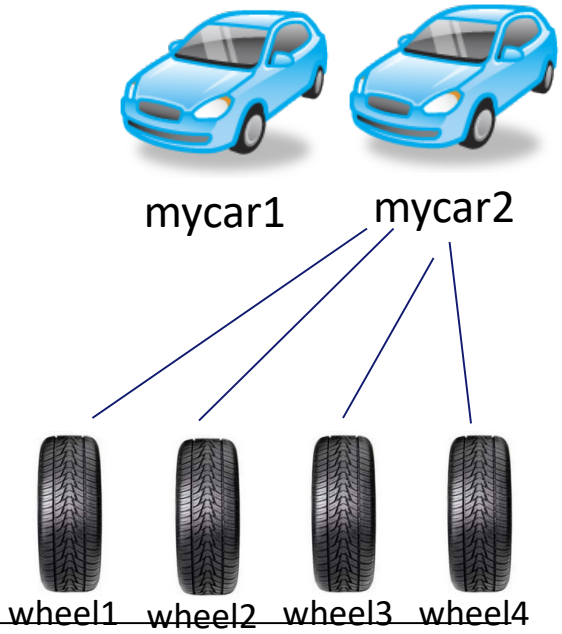


Any difference to association?

Yes! First, multiplicity must be 1 or 0..1. An instance can only have one owner.

But, isn't this equivalent to what we showed with associations?

Well, in this case...

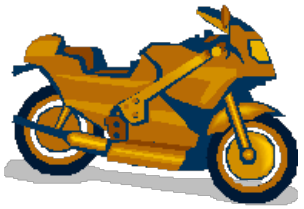


Relationships (3/6) - overview and intuition - Composition

Using composition...

mycar1

Ok for wheels to be part of mycar1 or mybike1



mybike1

wheel1 wheel2 wheel3 wheel4

wheel5 wheel6

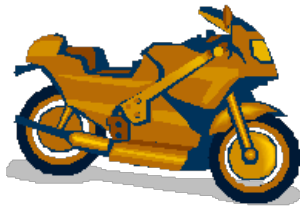
Relationships (3/6) - overview and intuition - Composition

Using composition...

Can mycar1 and mybike1 share the same wheels?

NO!
Not with composition!

mycar1



mybike1

wheel1 wheel2 wheel3 wheel4

Key concepts
- "No sharing" rule
- The owner is responsible for managing its parts, e.g. allocation and deallocation.

Relationships (3/6) - overview and intuition - Composition

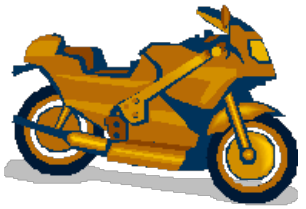
(Note the difference. The diamond is removed.)

Using associations...

Can mycar1 and mybike1 share the same wheels this time?

mycar1

Yes! Associations do not have a "no sharing" rule.



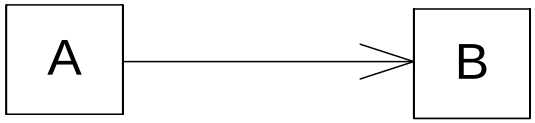
mybike1

However, in this case it is a strange model...

wheel1 wheel2 wheel3 wheel4

wheel5 wheel6

Relationships (4/6) - overview and intuition - Generalization

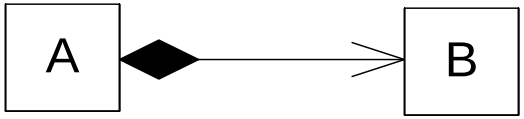


Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

Aggregation

Avoid it to avoid misunderstandings



Composition

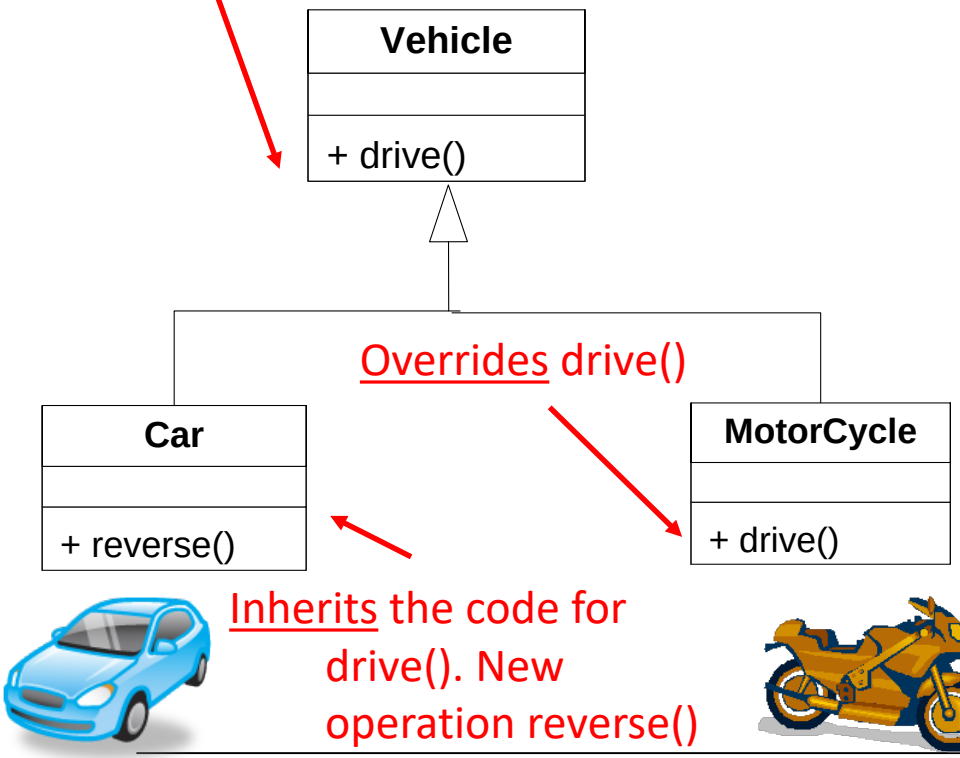
An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

Generalization

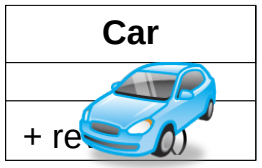
Relationships - (4/6) overview and intuition - Generalization

Class with code for the drive() operation

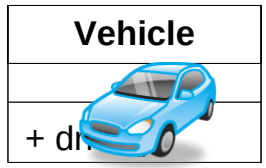
1. **Inheritance**
~ relation implementation



Visible Type: *Vehicle*.
Instance of: *MotorCycle*.
Can we drive()? Can we reverse()?



Visible Type: *Car*.
Instance of: *Car*
Can we drive()? Can we reverse()?



Visible Type: *Vehicle*.
Instance of: *Car*
Can we drive()? Can we reverse()?

An instance of a class can have many types
= (subtyping) polymorphism

2. **Subtyping**
~ relation on interfaces

static typing: safe substitution

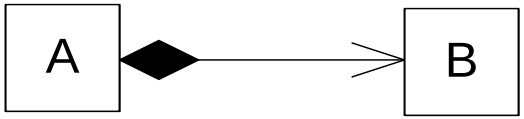
Relationships - (5/6) overview and intuition - Realization

Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

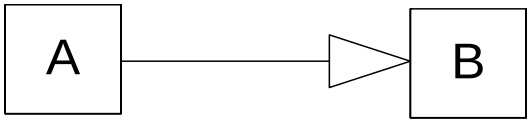
Aggregation

Avoid it to avoid misunderstandings



Composition

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

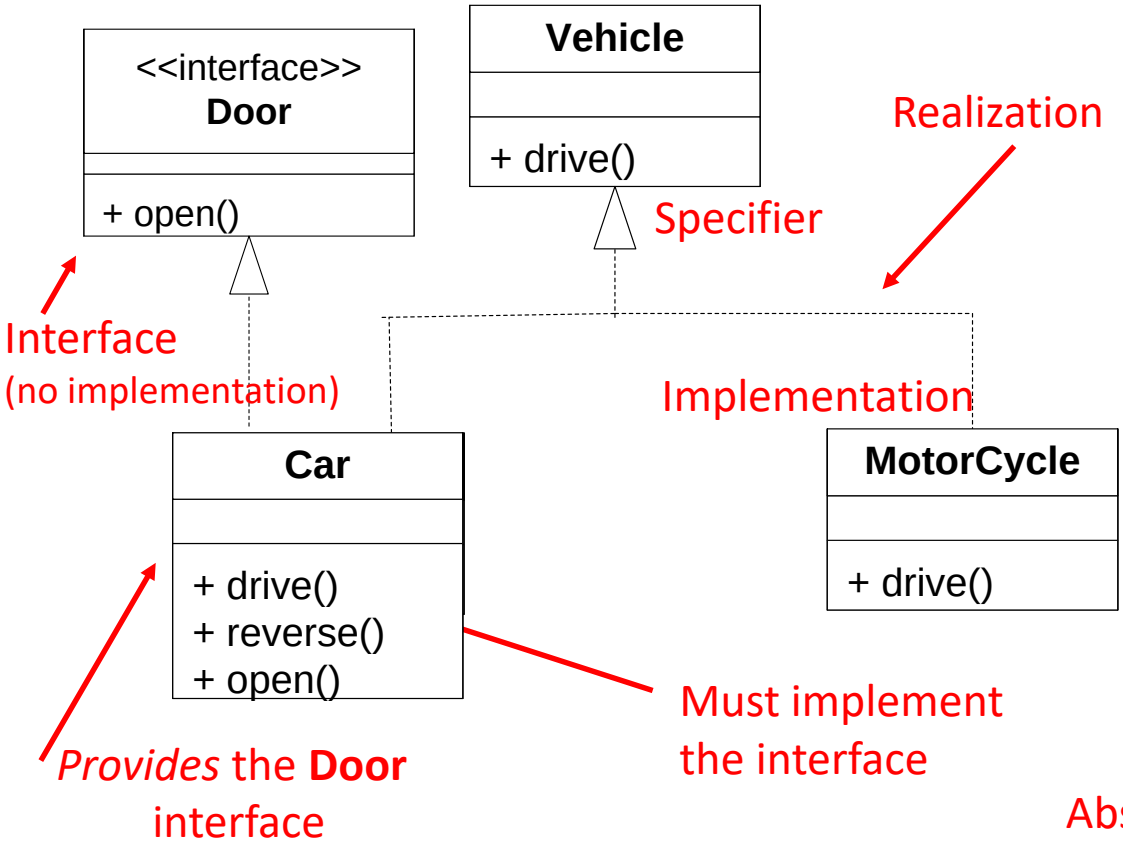


Generalization

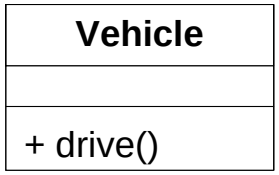
1) "A" inherits all properties and operations of "B".
2) An instance of "A" can be used where a instance of "B" is expected.

Realization

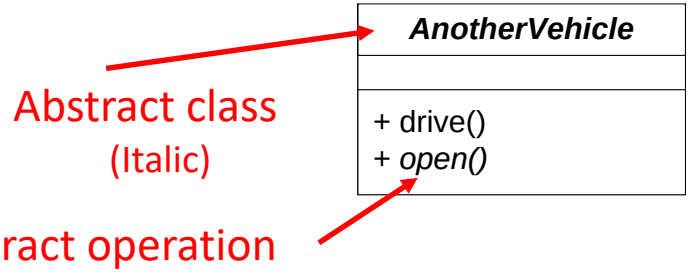
Relationships - (5/6) overview and intuition - Realization



Can we create an instance of Vehicle?

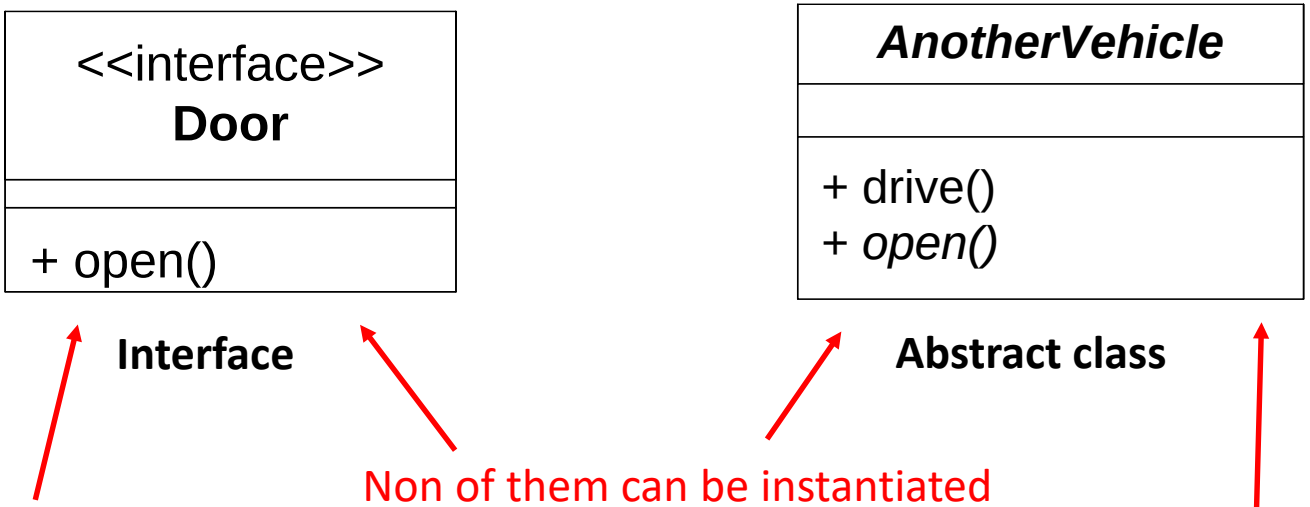


Can we create an instance of AnotherVehicle?



Relationships - (5/6) overview and intuition - Realization

What is the difference between an interface and an abstract class?



An abstract class with <u>only abstract operations</u> is conceptually the same as an interface

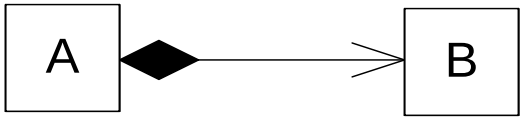
Relationships - (6/6) overview and intuition - Realization

Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

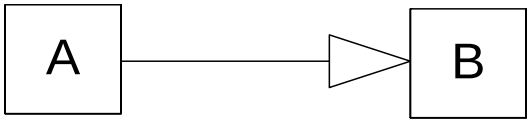
Aggregation

Avoid it to avoid misunderstandings



Composition

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.



Generalization

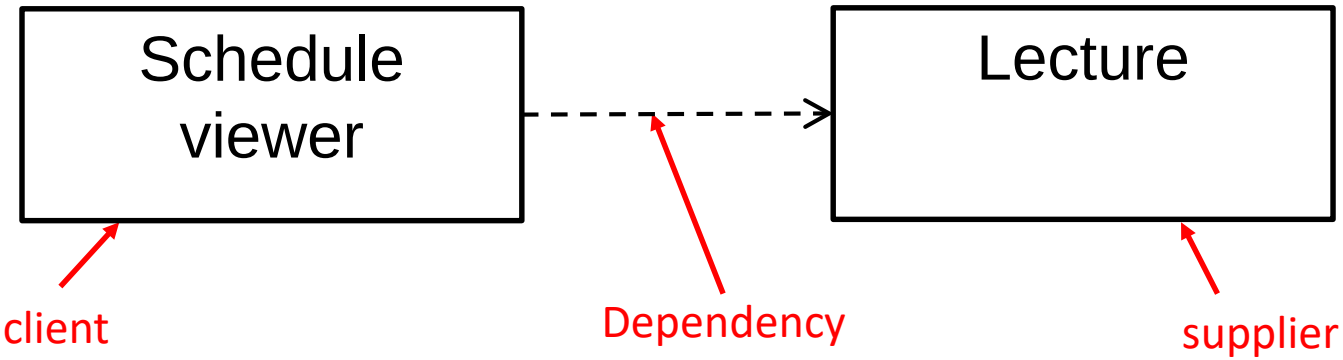
1) "A" inherits all properties and operations of "B".
2) An instance of "A" can be used where a instance of "B" is expected.

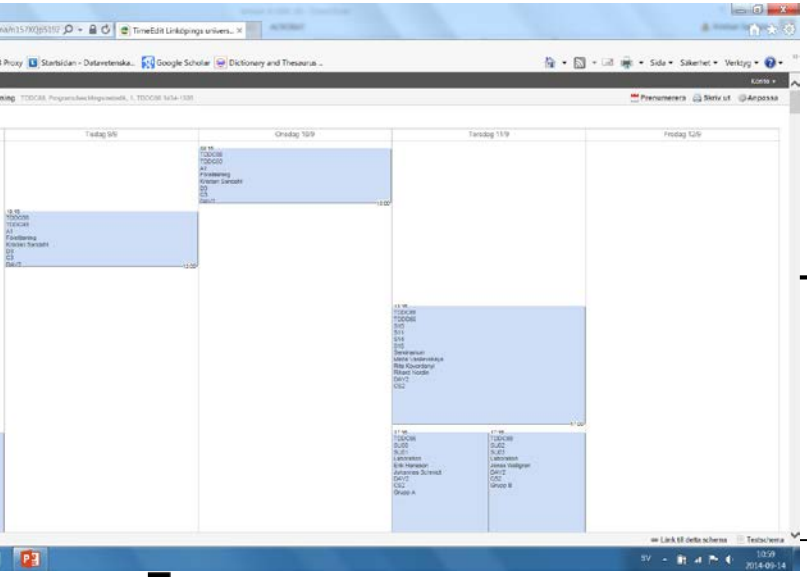
Realization

"A" provides an implementation of the interface specified by "B".

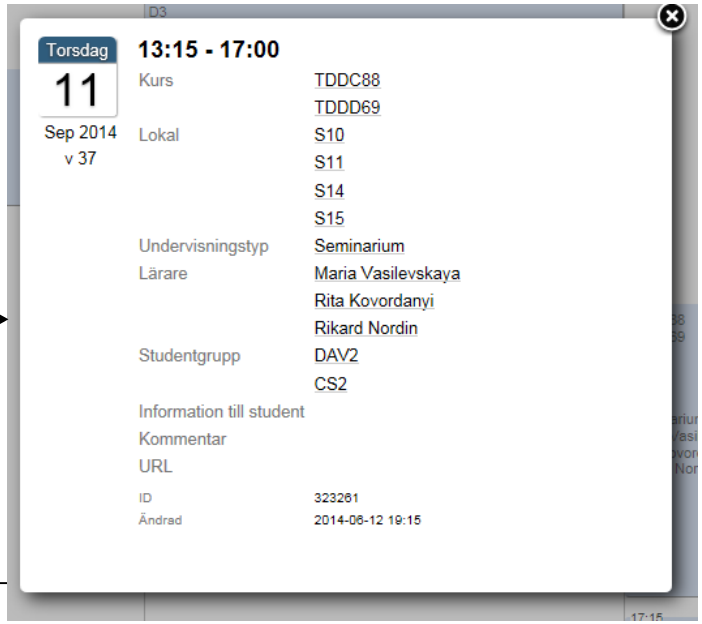
Dependency

Relationships - (6/6) overview and intuition - Dependency





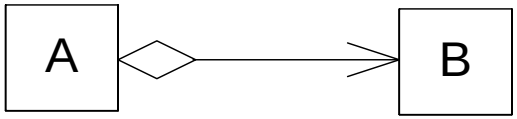
<<use>>



Relationships - overview and intuition

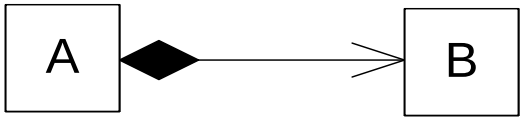
Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes



Aggregation

Avoid it to avoid misunderstandings



Composition

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

Generalization

1) "A" inherits all properties and operations of "B".
2) An instance of "A" can be used where a instance of "B" is expected.

Realization

"A" provides an implementation of the interface specified by "B".

Dependency

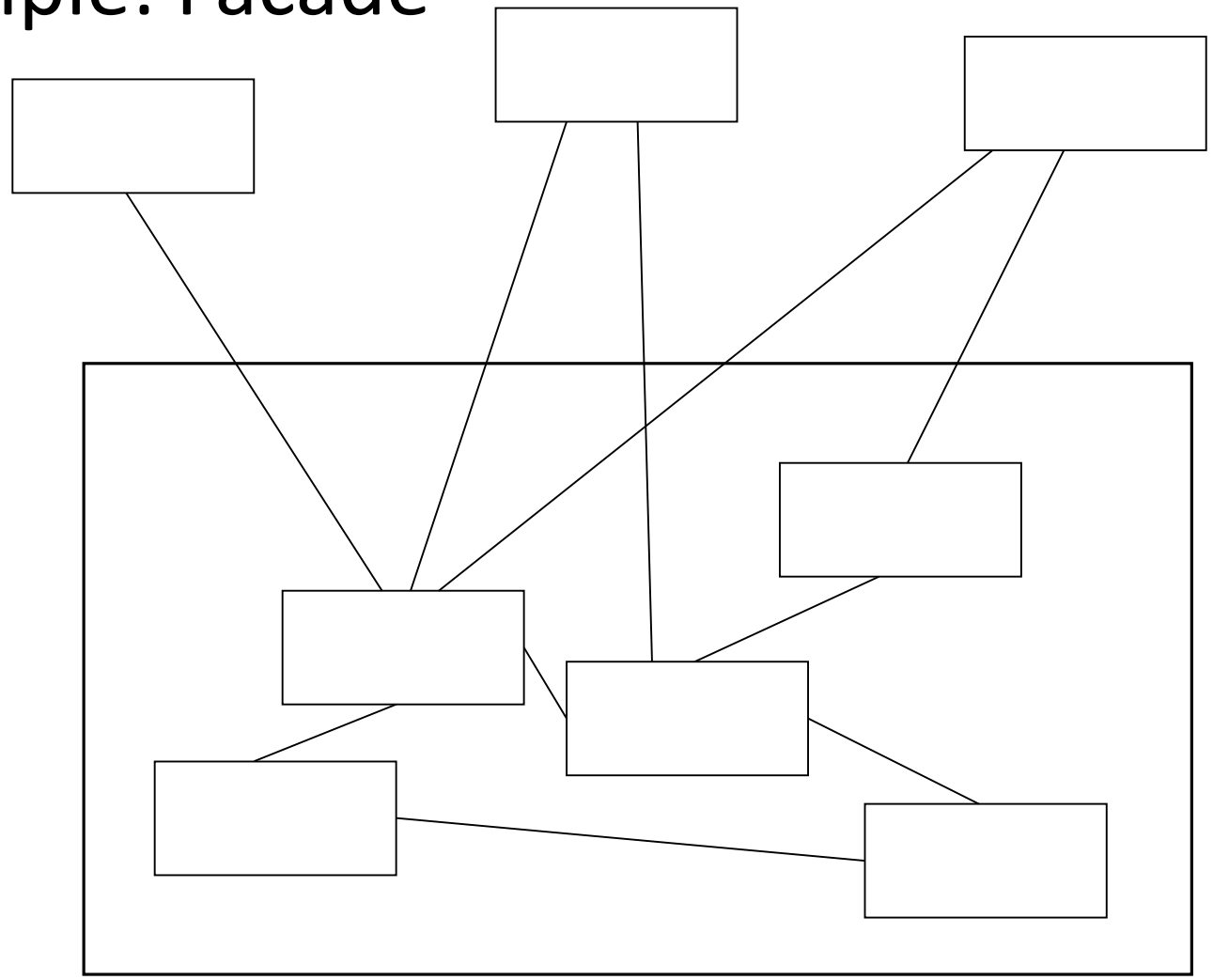
"A" is dependent on "B" if changes in the definition of "B" causes changes of "A".

Software Design Patterns

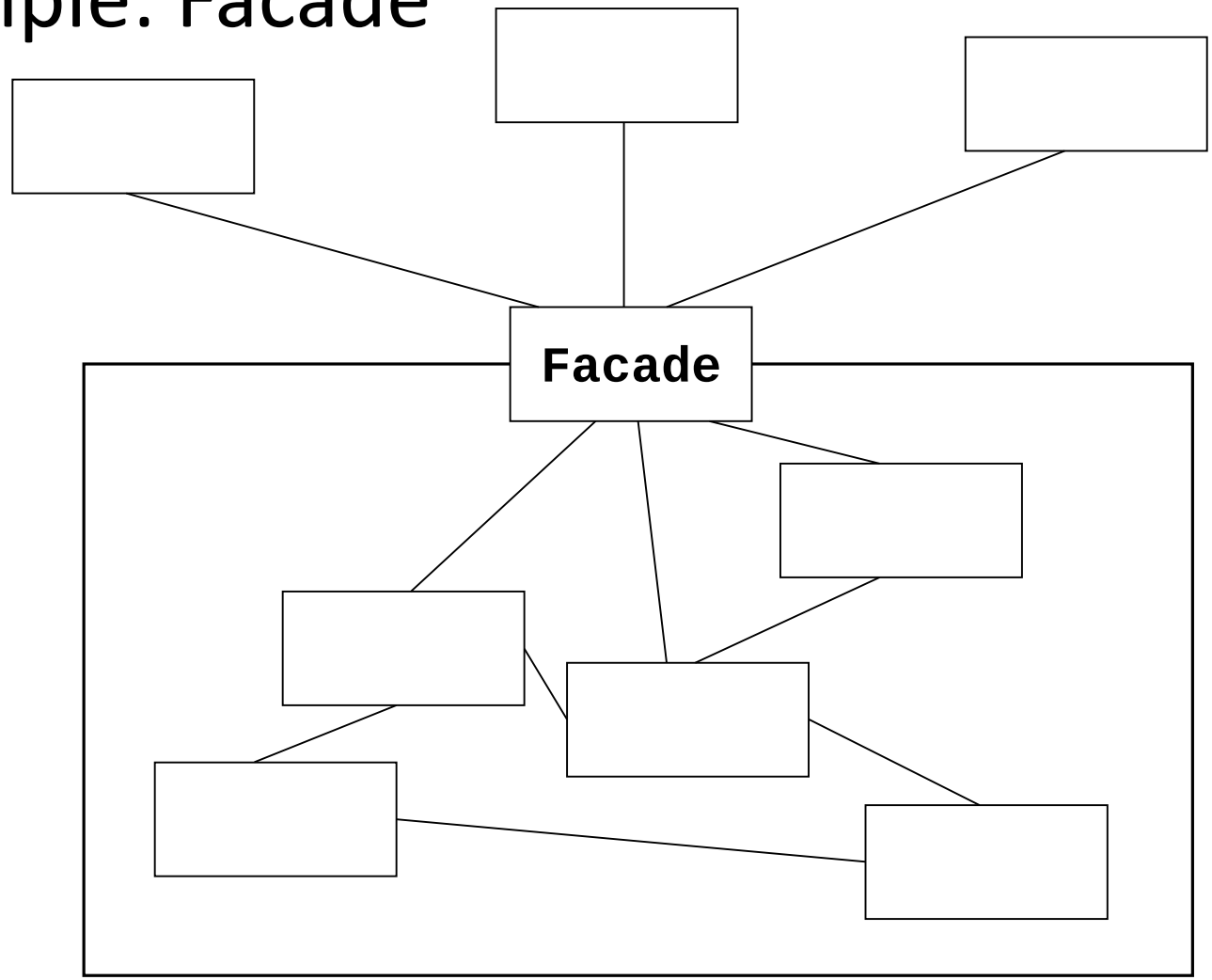
A Design Pattern is a standard solution for a standard design problem in a certain context.

Goal: reuse design information

Example: Facade

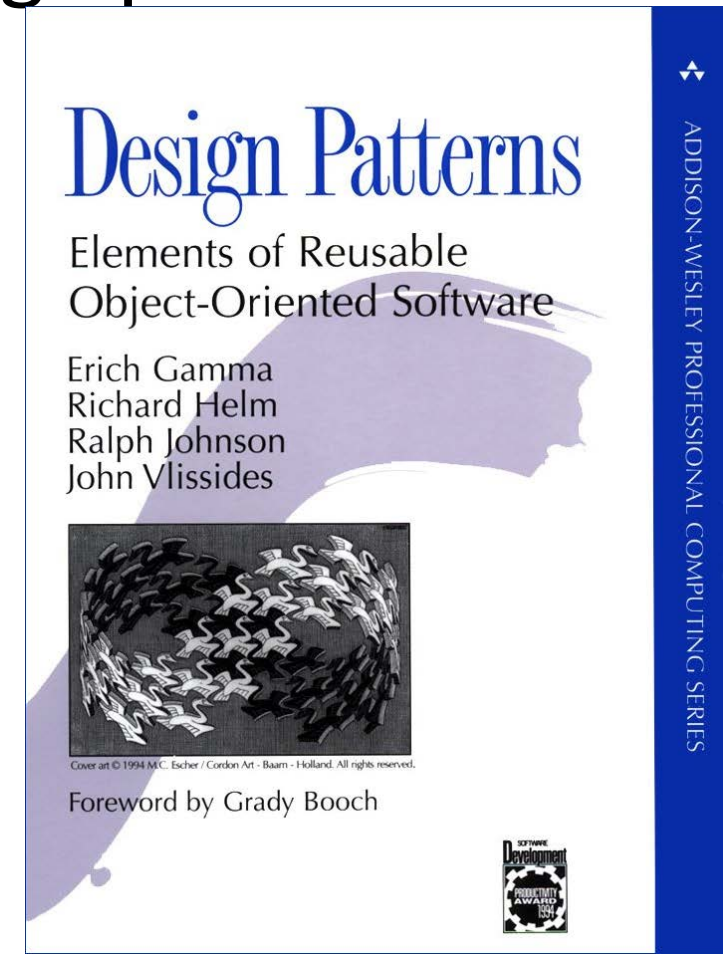


Example: Facade



How to describe design patterns?

- GoF book



Facade

Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. ... example ...

Facade

Applicability

Use the Facade pattern when:

- you want to provide a simple interface to a complex subsystem. This makes subsystems more reusable and easier to customize.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from other subsystems, thereby promoting subsystem independence and portability.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

Facade

Consequences

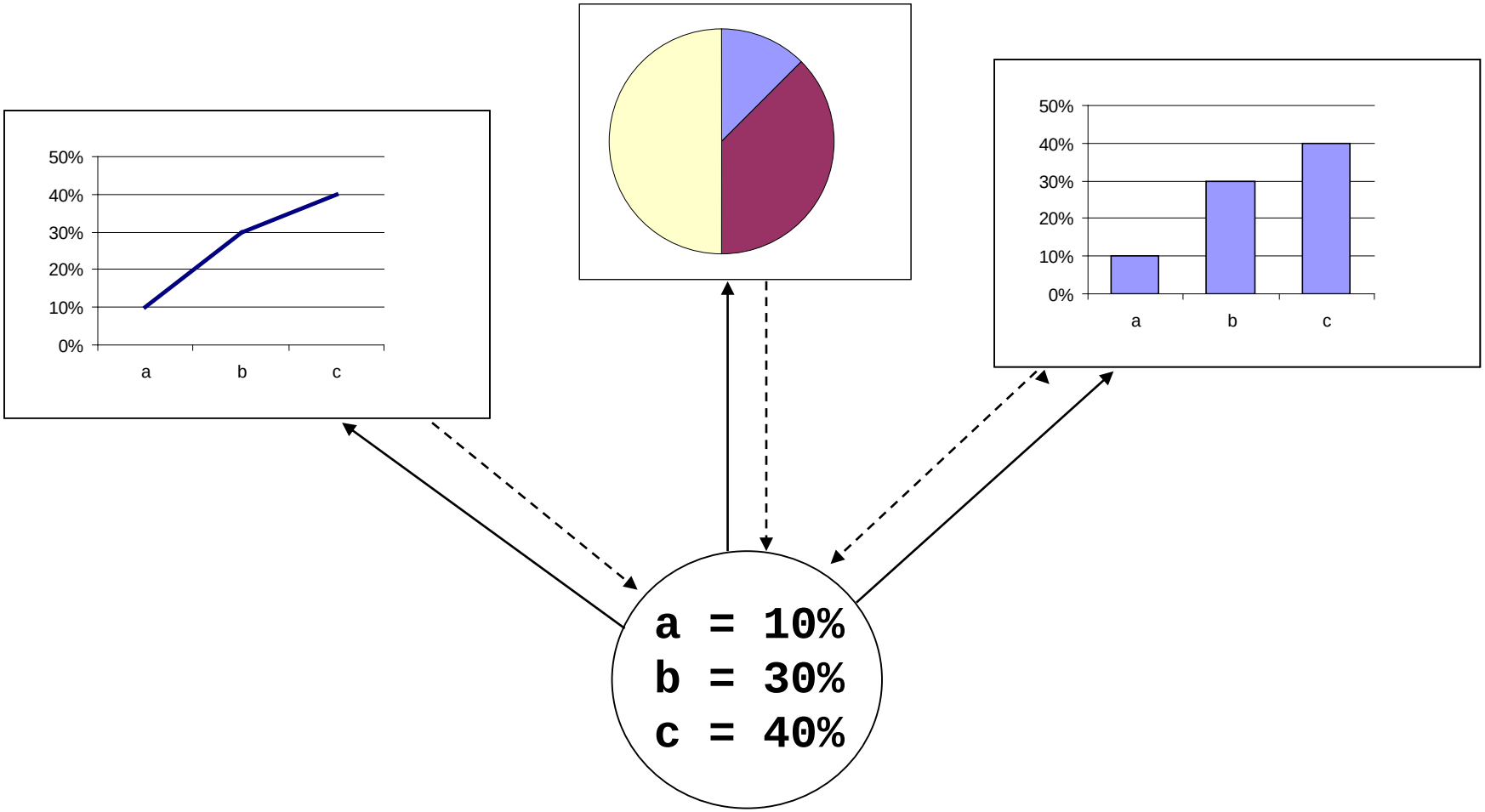
The Facade pattern offers the following benefits:

1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making subsystem easier to use.
2. It promotes weak coupling between subsystem and its clients. Weak coupling lets you vary the components of the subsystem without affecting its clients.
3. It doesn't prevent applications from using subsystem classes if they need to.

Facade

- **Structure**
- **Participants**
- **Collaborations**
- **Implementation**
- **Sample Code**
- **Known Uses**
- **Related Patterns**

Observer

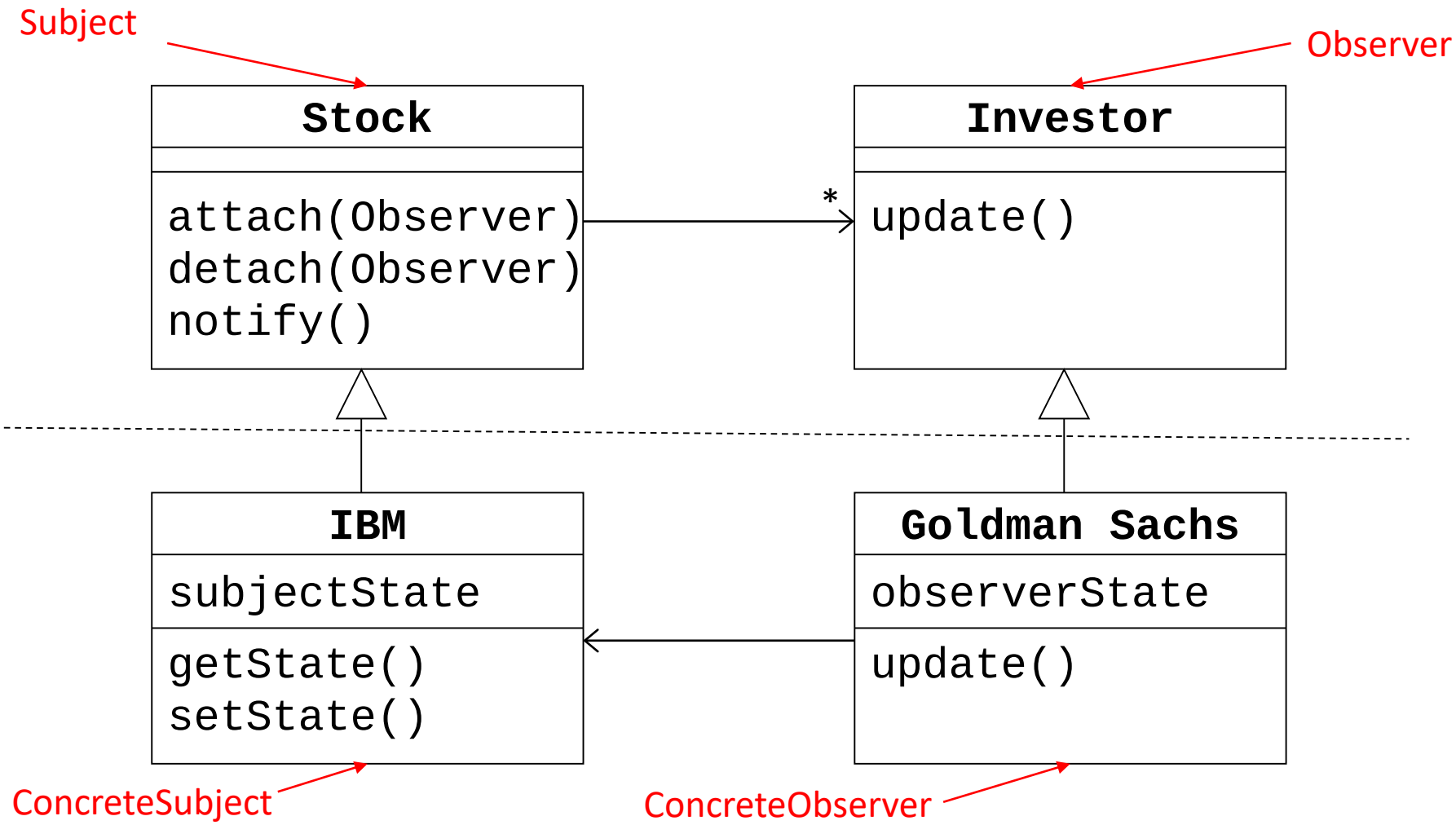


Observer

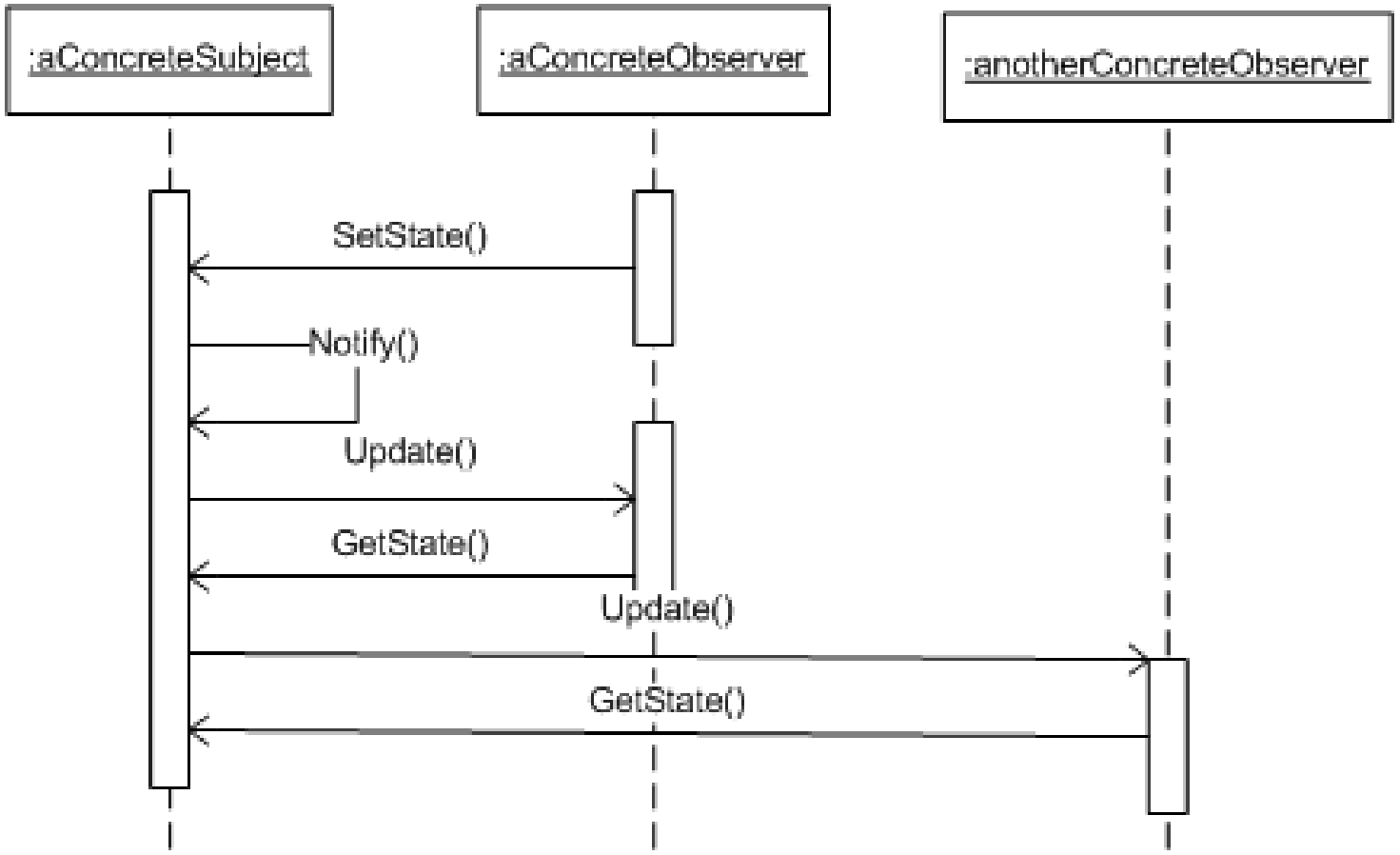
Applicability

- When an abstraction has two aspects, one dependent on the other.
- When a change to one object requires changing others.
- When an object should be able to notify other objects without making assumptions about who these objects are.

Observer, structure



Observer, collaborations



Observer, consequences

- Abstract coupling between Subject and Observer
- Support for broadcast communication
- Unexpected updates

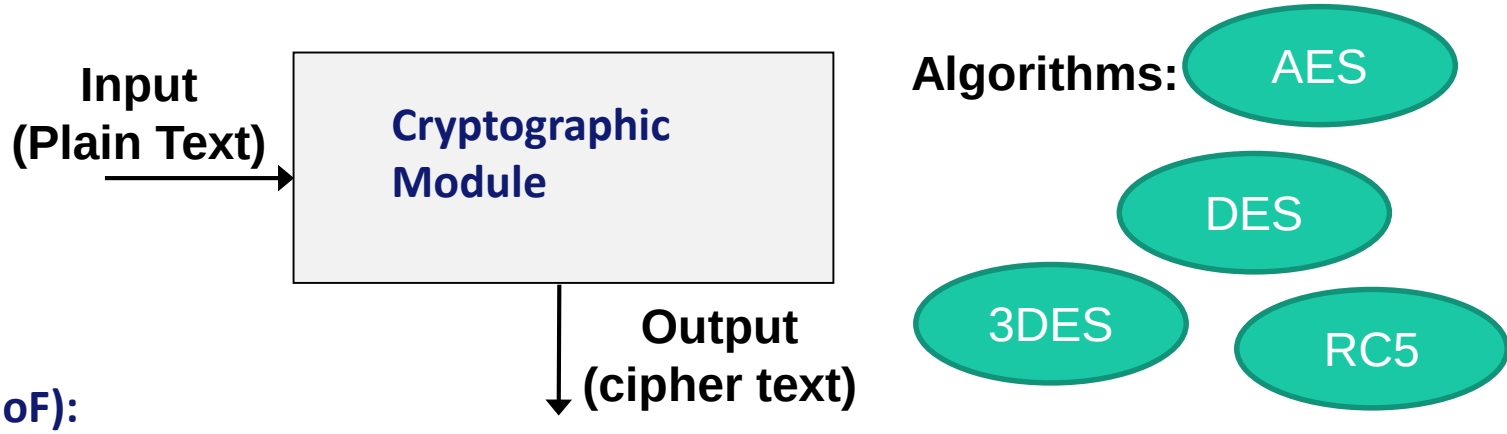
Strategy

Name: Strategy **Also known as:** Policy

Problem:

- Need to use different variants of the same algorithm in a class
- Different algorithms will be appropriate at different time.
- It is hard to add new algorithms and to change existing ones.

Example:

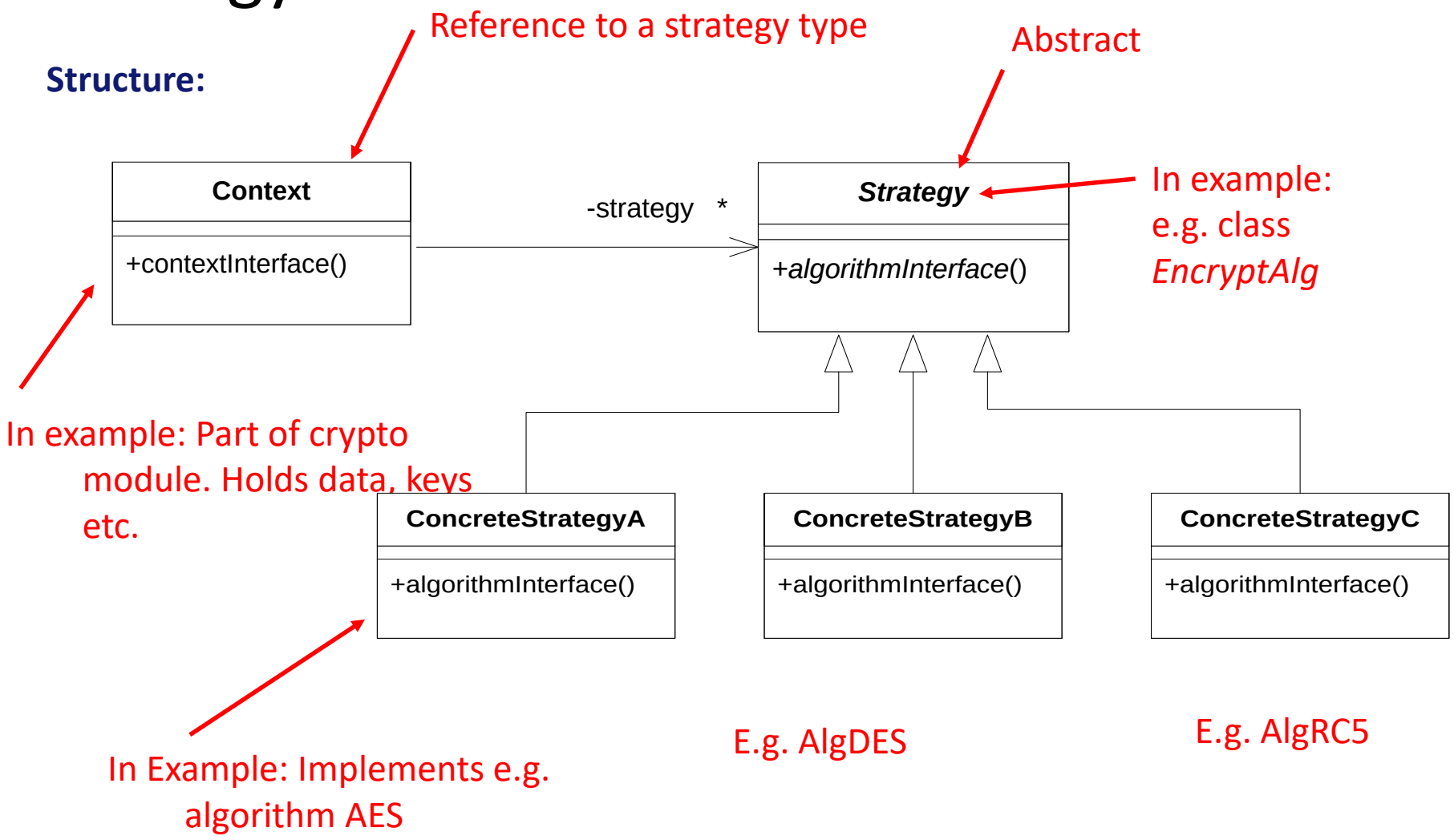


Intent (from GoF):

"Define a family of algorithms, encapsulate each one and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."

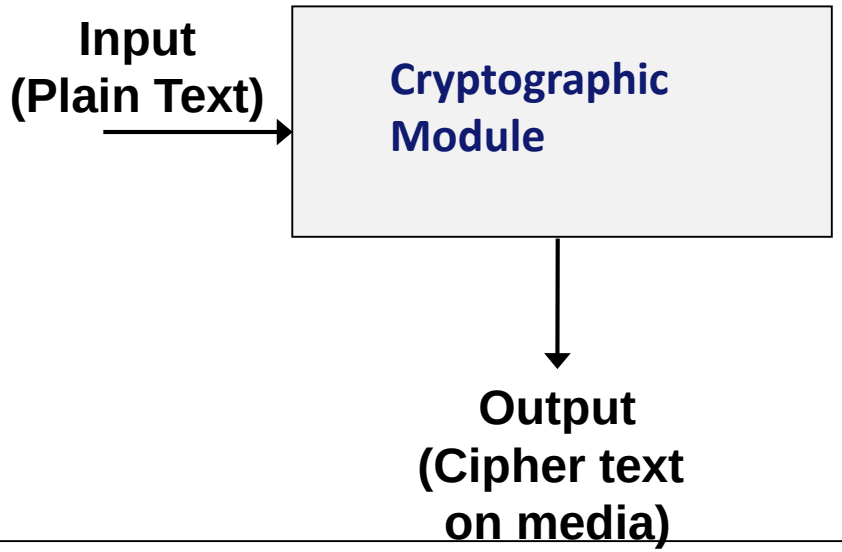
Strategy

Structure:

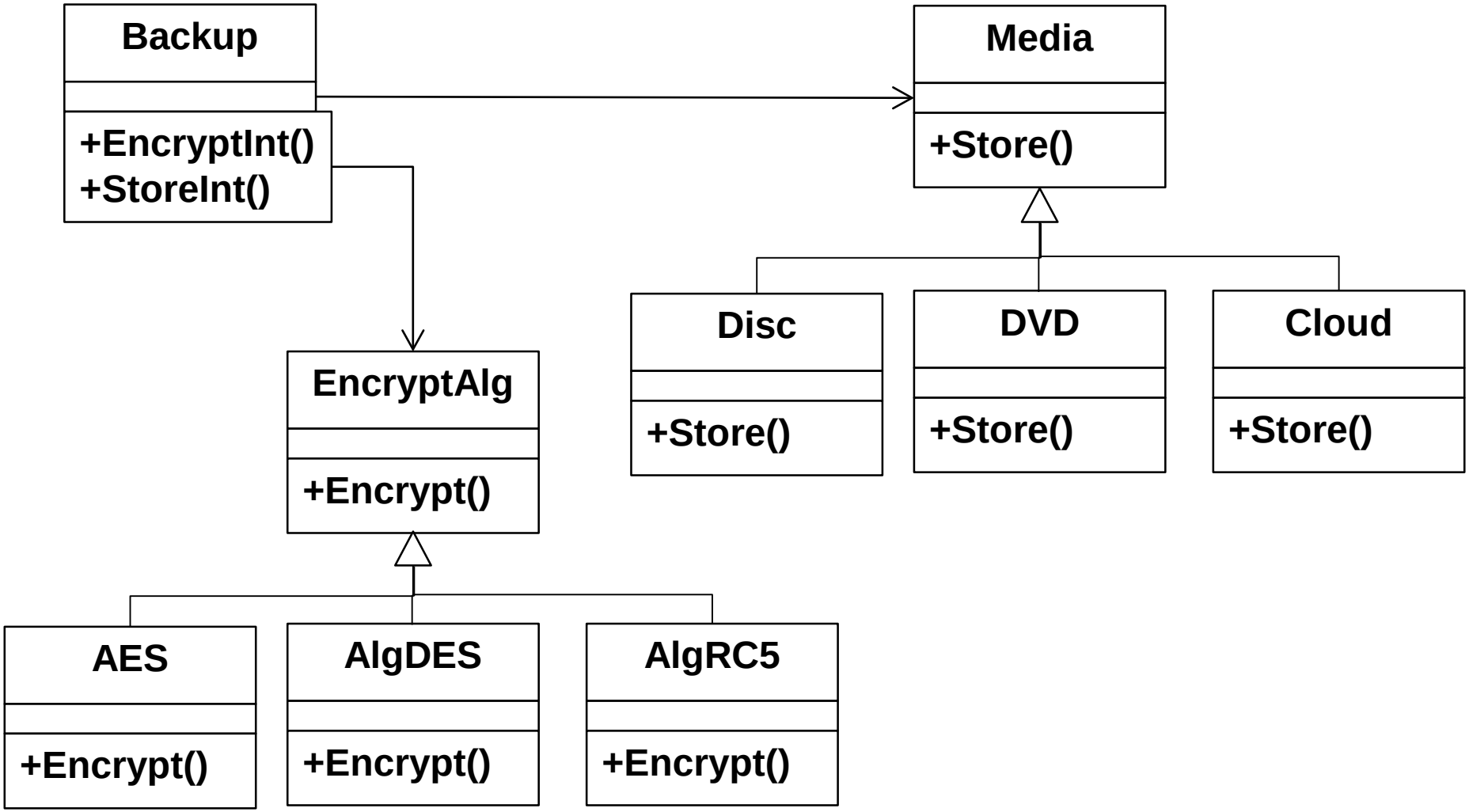


Strategy

- Suppose we add a new strategy:
- Storage media:
  - Disc
  - USB-stick
  - DVD
  - Cloud
    - ....



Two strategies



UML-OO/Kristian Sandahl

www.liu.se

LINKÖPING UNIVERSITY