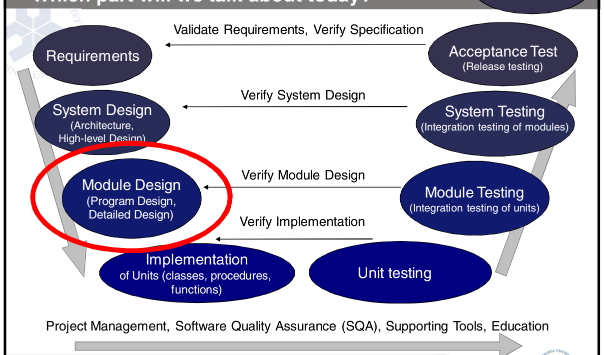
Introduction to UML and Design Patterns

DF14900 Software Engineering
CUGS
2011

Christoph Kessler and Kristian Sandahl
Department of Computer and Information Science
Linköping University, Sweden

A Software Life-cycle Model… Which part will we talk about today?

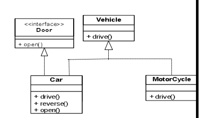


The goals of module design

- Provide the expected function
- Prepare for change:
  - Separation of concern
  - Testability
  - Understandability
- Contribute to quality, eg:
  - Performance
  - Usability
  - Reliability
  - …

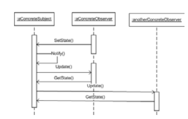
Agenda

Part I:
Structural Modeling with UML
Esp., Classes and Objects



Part II:
Short Introduction to Design Patterns

Part III:
Behavioral Modeling with UML
State Machines, Sequence Diagrams, Use case diagrams, Activity Diagrams



Part I

Modeling Structure with UML

Classes and Objects

Modeling as a Design Technique

- Testing a physical entity before building it
- Communication with customers
- Visualization
- Reduction of complexity

- Models **supplement** natural language
- Models support understanding, design, documentation
- Creating a model forces you to take necessary design decisions
- **UML** is now the standard notation for modeling software.

Literature on UML

- Official standard documents by OMG: www.omg.org, www.uml.org
- Current version is UML 2.0 (2004/2005)
- OMG documents: *UML Infrastructure*, *UML Superstructure*
- Books:
  - Pfleeger: *Software Engineering* 3rd ed., 2005 (mostly Chapter 6)
  - Rumbaugh, Jacobson, Booch: *The Unified Modeling Language Reference Manual*, Second Edition, Addison-Wesley 2005
  - Blaha, Rumbaugh: *Object-Oriented Modeling and Design with UML*, Second Edition, Prentice-Hall, 2005.
  - Stevens, Pooley: *Using UML: Software Engineering with Objects and Components*, 2nd edition. Addison-Wesley, 2006
  - And many others…

UML: Different diagram types for different views of software

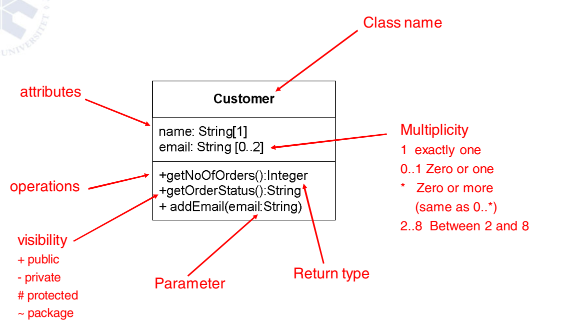
Modeling (logical) structure of software:
- Static view: Class diagram
- Design view: Structure diagr., collaboration d., component d.
- Use case view: Use case diagram

Modeling behavior of software:
- Activity view: Activity diagram
- State machine view: State machine diagram
- Interaction view: Sequence diagram, communication diagram

Modeling physical structure of software
- Deployment view: Deployment diagram

Modeling the model, and extending UML itself
- Model management view: Package Diagram
- Profiles

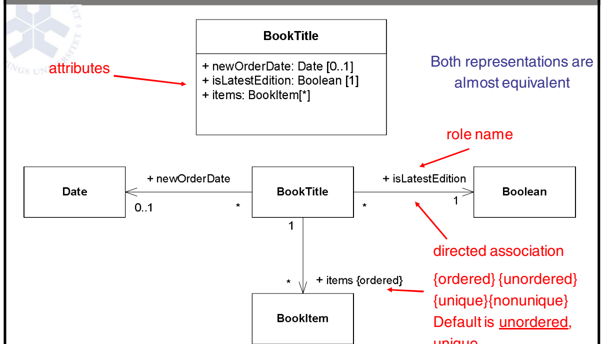
A Single Class



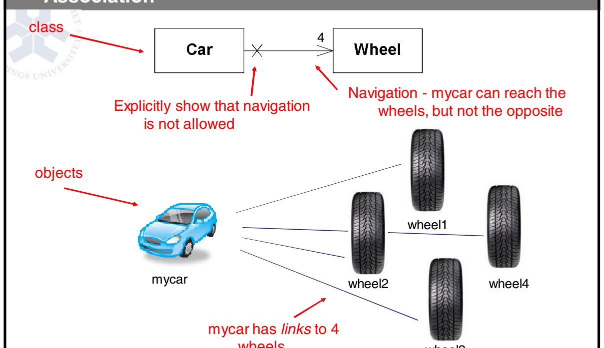
Relationships (1/6) - overview and intuition - Association

Association
(with navigability)

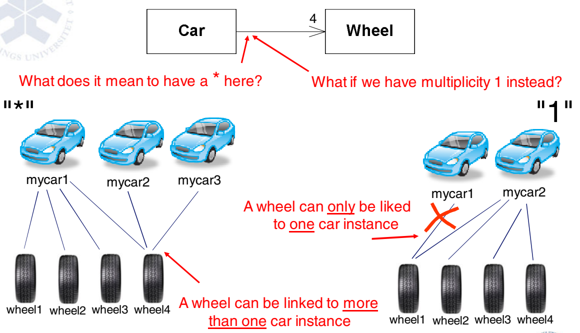
Relationships (1/6) - overview and intuition - Association



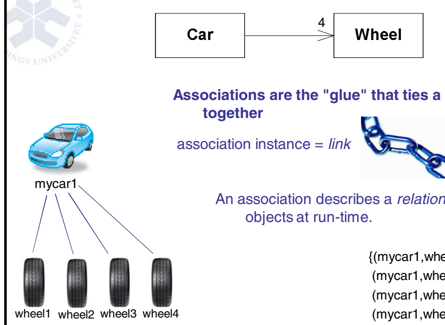
Relationships (1/6) - overview and intuition - Association



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



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



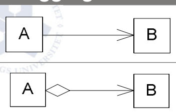
Associations are the "glue" that ties a system together

association instance = *link*

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

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

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

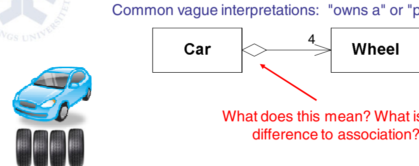


Association (with navigability) — *"A" has a reference(s) to instance(s) of "B". Alternative: attributes*

Aggregation

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

A major source of confusion
Common vague interpretations: "owns a" or "part of"



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

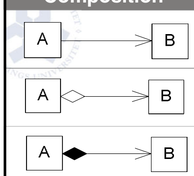
Vague definitions → Inconsistency and misunderstandings

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



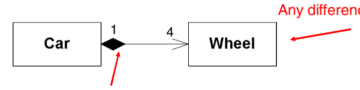
Association (with navigability) — *"A" has a reference(s) to instance(s) of "B". Alternative: attributes*

Aggregation — Avoid it to avoid misunderstandings

Composition

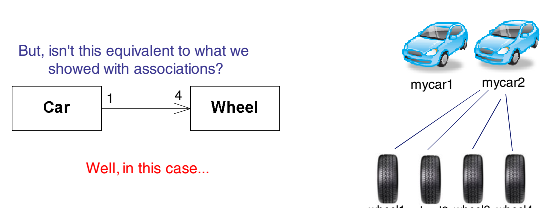
Relationships (3/6) - overview and intuition - Composition

Any difference to association?

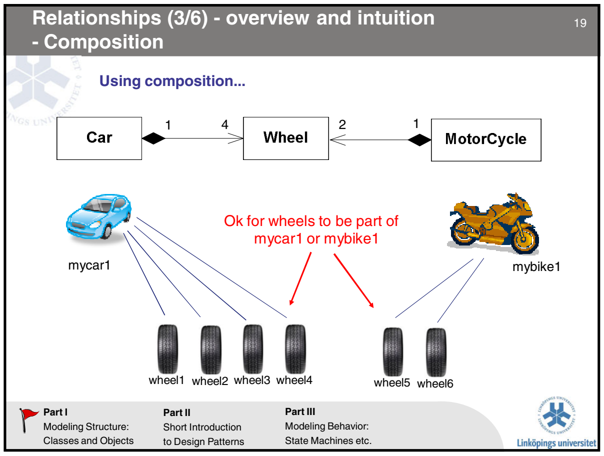


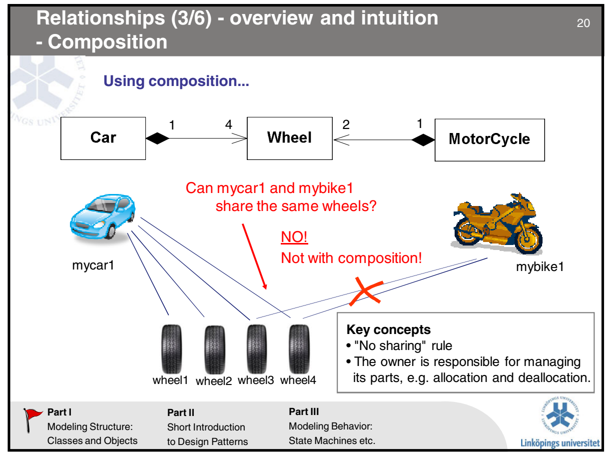
Yes! First, multiplicity must be 1 or 0..1. An instance can only have one owner.

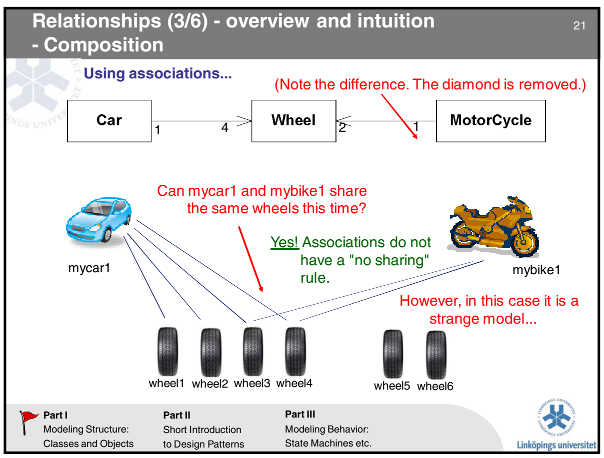
But, isn't this equivalent to what we showed with associations?

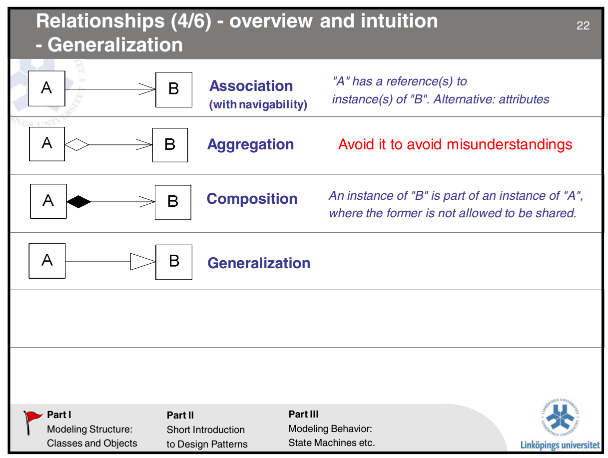


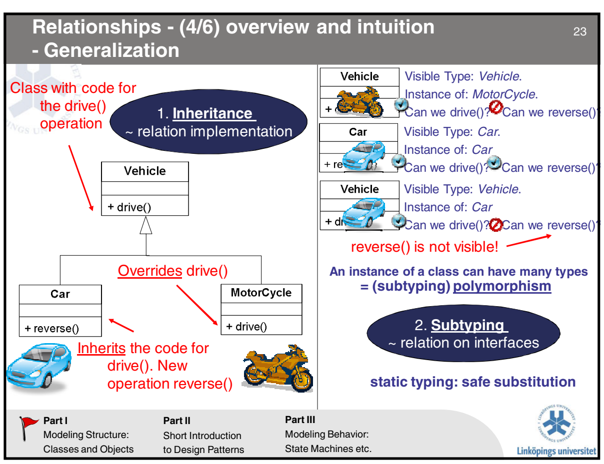
Well, in this case...

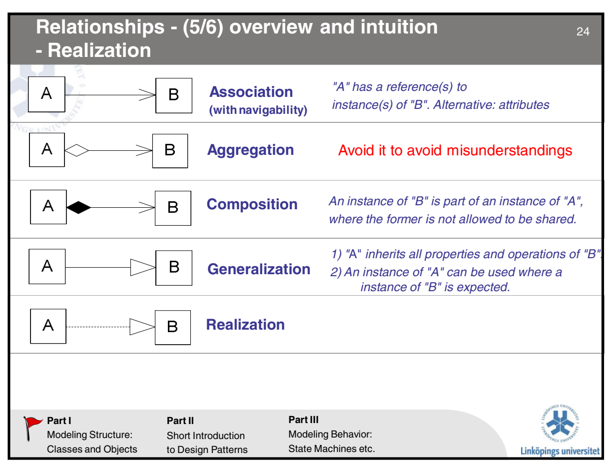

Relationships (3/6) - overview and intuition
- Composition
19
Using composition...
Car
1
4
Wheel
2
1
MotorCycle
Ok for wheels to be part of
mycar1 or mybike1
mycar1
mybike1
wheel1 wheel2 wheel3 wheel4
wheel5 wheel6
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet

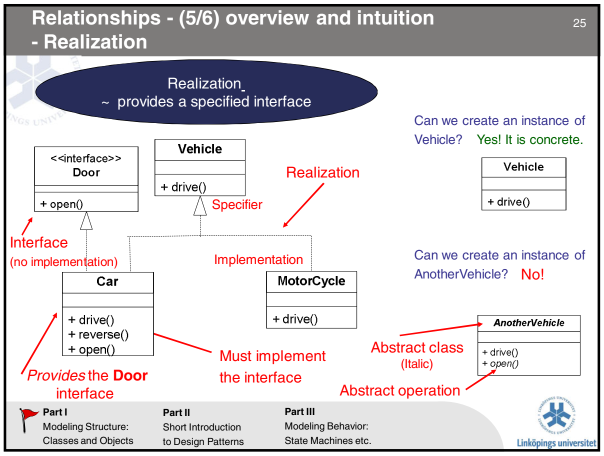

Relationships (3/6) - overview and intuition
- Composition
20
Using composition...
Car
1
4
Wheel
2
1
MotorCycle
Can mycar1 and mybike1
share the same wheels?
NO!
Not with composition!
mycar1
mybike1
wheel1 wheel2 wheel3 wheel4
Key concepts
• "No sharing" rule
• The owner is responsible for managing
its parts, e.g. allocation and deallocation.
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet

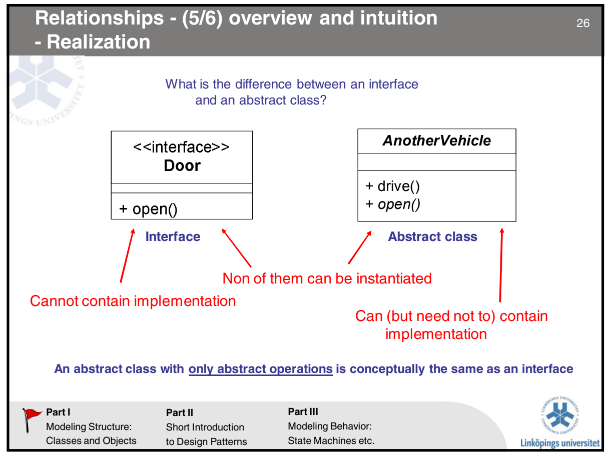

Relationships (3/6) - overview and intuition
- Composition
21
Using associations...
(Note the difference. The diamond is removed.)
Car
1
4
Wheel
2
1
MotorCycle
Can mycar1 and mybike1 share
the same wheels this time?
Yes! Associations do not
have a "no sharing"
rule.
However, in this case it is a
strange model...
mycar1
mybike1
wheel1 wheel2 wheel3 wheel4
wheel5 wheel6
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet

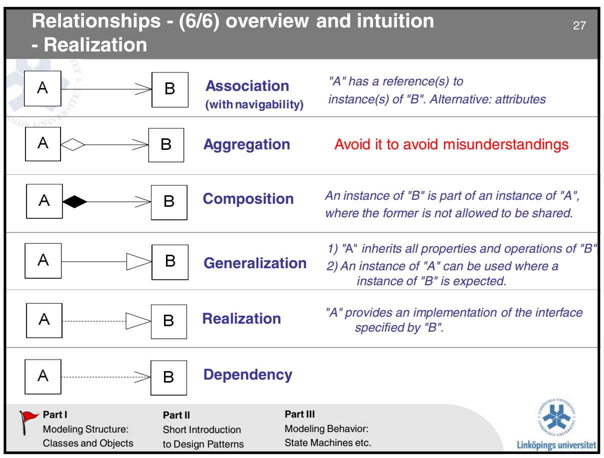

Relationships (4/6) - overview and intuition
- Generalization
22
A
B
Association
(with navigability)
"A" has a reference(s) to
instance(s) of "B". Alternative: attributes
A
B
Aggregation
Avoid it to avoid misunderstandings
A
B
Composition
An instance of "B" is part of an instance of "A",
where the former is not allowed to be shared.
A
B
Generalization
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet

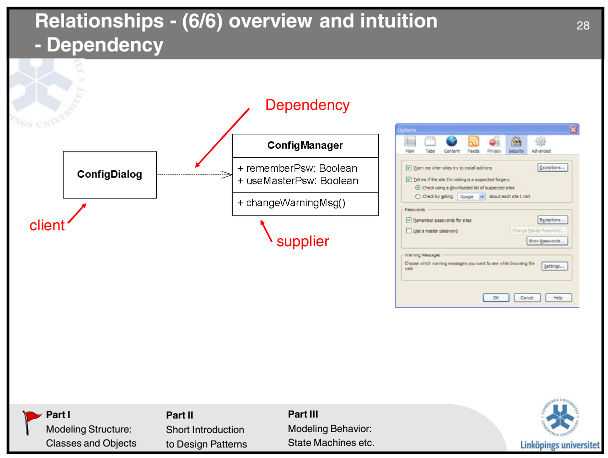

Relationships - (4/6) overview and intuition
- Generalization
23
Class with code for
the drive()
operation
1. Inheritance
~ relation implementation
Vehicle
+ drive()
Overrides drive()
Car
+ reverse()
MotorCycle
+ drive()
Inherits the code for
drive(). New
operation reverse()
Vehicle
Visible Type: Vehicle.
Instance of: MotorCycle.
Can we drive()? Can we reverse()
Car
Visible Type: Car.
Instance of: Car
Can we drive()? Can we reverse()
Vehicle
Visible Type: Vehicle.
Instance of: Car
Can we drive()? Can we reverse()
reverse() is not visible!
An instance of a class can have many types
= (subtyping) polymorphism
2. Subtyping
~ relation on interfaces
static typing: safe substitution
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet

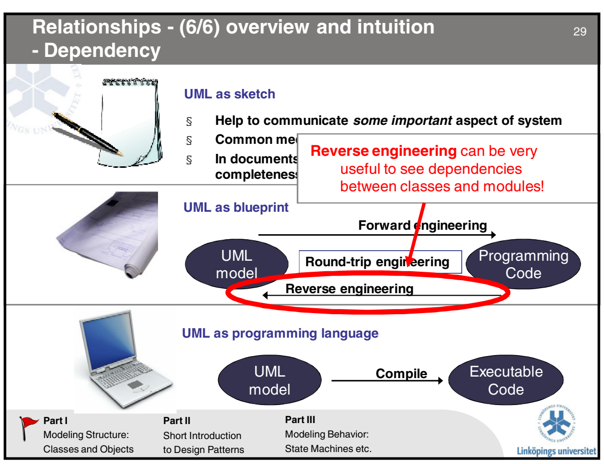

Relationships - (5/6) overview and intuition
- Realization
24
A
B
Association
(with navigability)
"A" has a reference(s) to
instance(s) of "B". Alternative: attributes
A
B
Aggregation
Avoid it to avoid misunderstandings
A
B
Composition
An instance of "B" is part of an instance of "A",
where the former is not allowed to be shared.
A
B
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.
A
B
Realization
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet

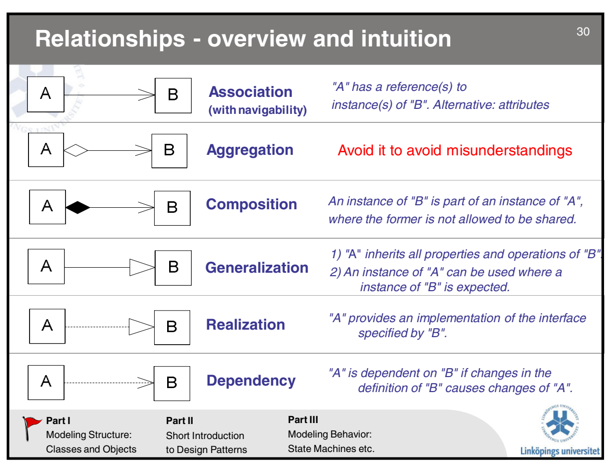

Relationships - (5/6) overview and intuition - Realization
25
Realization
~ provides a specified interface
Can we create an instance of Vehicle? Yes! It is concrete.
<<interface>>
Door
+ open()
Vehicle
+ drive()
Realization
Specifier
Vehicle
+ drive()
Interface
(no implementation)
Implementation
Car
+ drive()
+ reverse()
+ open()
MotorCycle
+ drive()
Can we create an instance of AnotherVehicle? No!
AnotherVehicle
+ drive()
+ open()
Must implement the interface
Abstract class
(Italic)
Provides the Door interface
Abstract operation
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet


Relationships - (5/6) overview and intuition - Realization
26
What is the difference between an interface and an abstract class?
<<interface>>
Door
+ open()
AnotherVehicle
+ drive()
+ open()
Interface
Abstract class
Non of them can be instantiated
Cannot contain implementation
Can (but need not to) contain implementation
An abstract class with only abstract operations is conceptually the same as an interface
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet

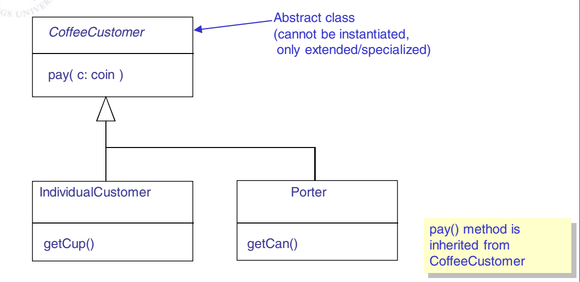

Relationships - (6/6) overview and intuition - Realization
27
A B Association (with navigability) "A" has a reference(s) to instance(s) of "B". Alternative: attributes
A B Aggregation Avoid it to avoid misunderstandings
A B Composition An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.
A B 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.
A B Realization "A" provides an implementation of the interface specified by "B".
A B Dependency
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet


Relationships - (6/6) overview and intuition - Dependency
28
Dependency
ConfigDialog
ConfigManager
+ rememberPsw: Boolean
+ useMasterPsw: Boolean
+ changeWarningMsg()
client
supplier
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet

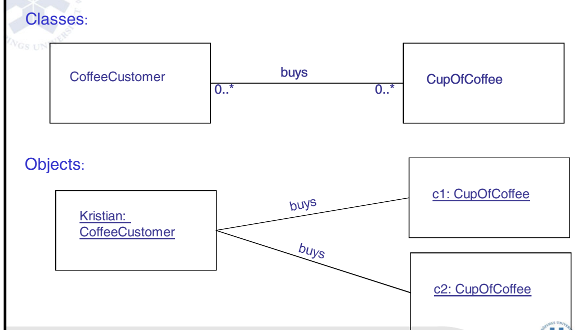

Relationships - (6/6) overview and intuition - Dependency
29
UML as sketch
§ Help to communicate some important aspect of system
§ Common me
§ In documents
completeness
Reverse engineering can be very useful to see dependencies between classes and modules!
UML as blueprint
Forward engineering
UML model
Round-trip engineering
Programming Code
Reverse engineering
UML as programming language
UML model
Compile
Executable Code
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet


Relationships - overview and intuition
30
A B Association (with navigability) "A" has a reference(s) to instance(s) of "B". Alternative: attributes
A B Aggregation Avoid it to avoid misunderstandings
A B Composition An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.
A B 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.
A B Realization "A" provides an implementation of the interface specified by "B".
A B Dependency "A" is dependent on "B" if changes in the definition of "B" causes changes of "A".
Part I
Modeling Structure:
Classes and Objects
Part II
Short Introduction
to Design Patterns
Part III
Modeling Behavior:
State Machines etc.
Linköpings universitet

Class model with inheritance and abstract classes



Classes and objects



Reasoning about an arbitrary object

Like this:

...or simply like this:

Part II

Short Introduction to Design Patterns

Christopher Alexander

TAPESTRY OF LIGHT AND DARK

Create alternating areas of light and dark throughout the building, in such a way that **people naturally walk toward the light,** whenever they are going to important places: seats, entrances, stairs, passages, places of special beauty, and make other areas darker, to increase the contrast.

Software Design Patterns

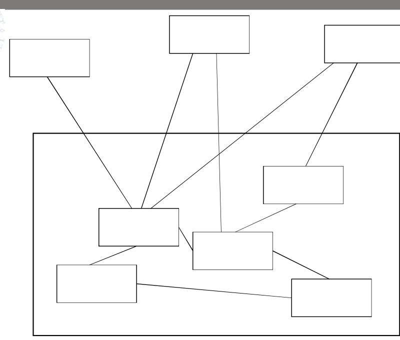
A Design Pattern is a standard solution for a standard design problem in a certain context.

Goal: reuse design information

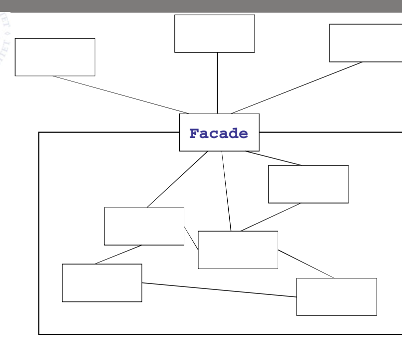
*"A **pattern** involves a general description of a recurring solution to a recurring problem with various goals and constraints. It identifies more than a solution, it also explains why the solution is needed."* (James Coplien)

"... describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Christopher Alexander)

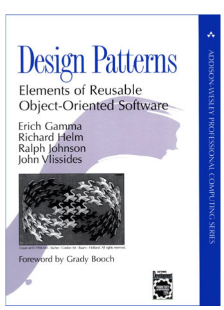
Example: Facade



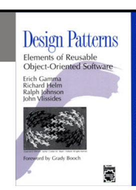
Example: Facade



How to describe design patterns?



Pattern Catalog in Gamma et al. 1996:



The GoF book describes each pattern using the following attributes:

- The ***name*** to describes the pattern, its solutions and consequences in a word or two
- The ***problem*** describes when to apply the pattern: *intent, motivation, applicability*
- The ***solution*** describes the elements that make up the *design (structure with participants)*, their *relationships*, *responsibilities*, and *collaborations/interactions*
- The ***consequences*** are the results and trade-offs in applying the pattern

Also: *implementation notes, known uses, related patterns.*

All ***examples*** in C++ and Smalltalk

Remark: Patterns exist also beyond the OO world…

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.

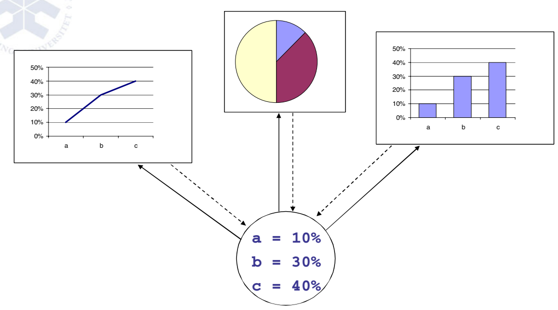
Facade

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.

Observer

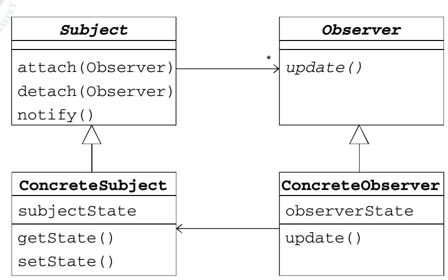


Observer

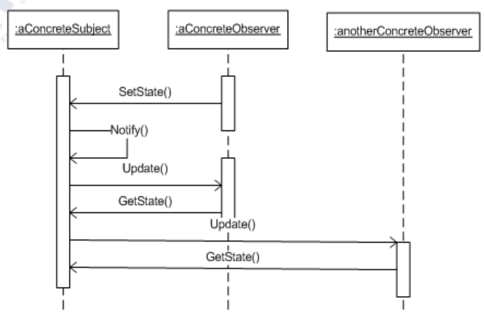
Applicability

- 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

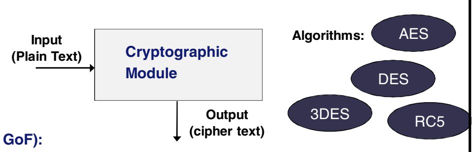
Strategy

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:

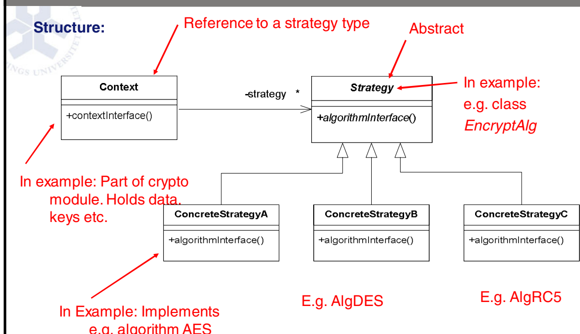


Intent (from GoF):

"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:



Design Pattern Space [Gamma et al.'96]

Creational Patterns	Structural Patterns	Behavioral Patterns
Factory Method	Adapter (class based)	Interpreter
Abstract factory	Adapter (object-based)	Template method
Builder	Bridge	Chain of Responsibility
Prototype	Composite	Command Iterator
Singleton	Decorator	Mediator Memento
	Facade Flyweight	Observer State
	Proxy	Strategy Visitor

Creational patterns

Deal with initializing and configuring of classes and objects

Structural patterns

Deal with decoupling interface and implementation of classes and objects

Behavioral patterns

Deal with dynamic interactions among societies of classes and objects

Part III

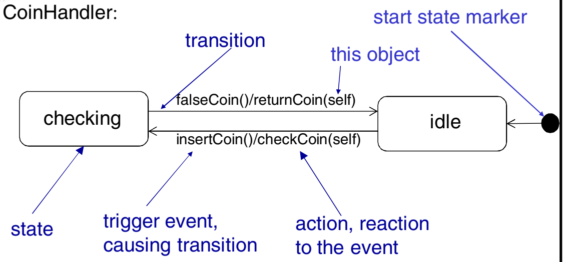
Modeling Behavior in UML: State Machines etc.

Further UML Features

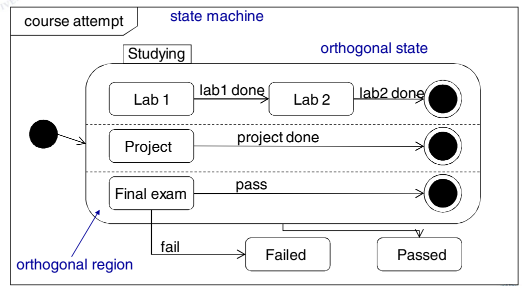
State machine diagram

Can formally describe protocols

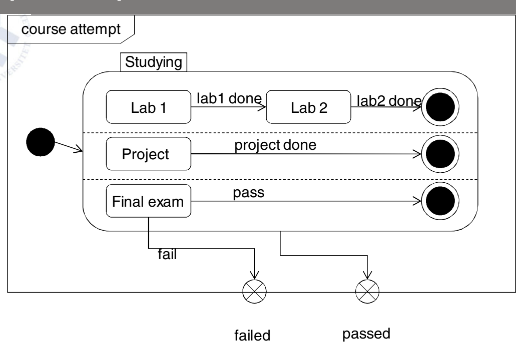
For class CoinHandler:



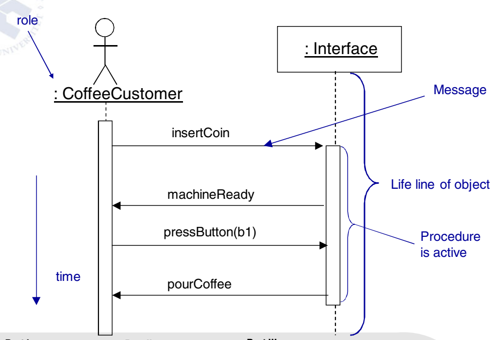
Orthogonal, composite state



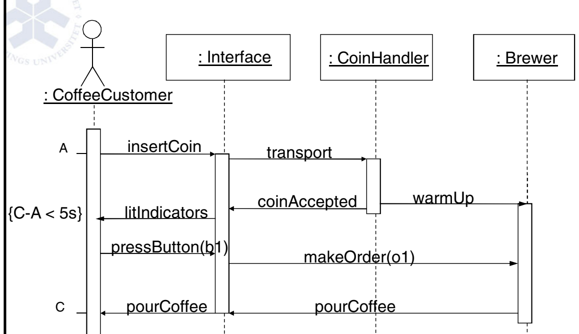
Explicit exit points



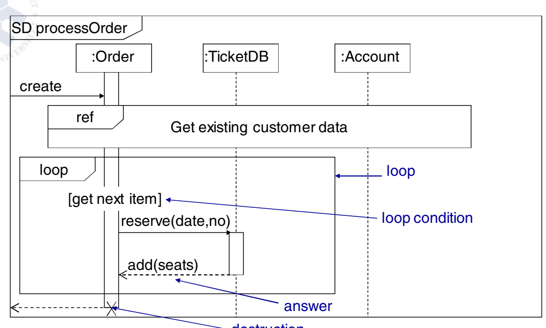
Sequence diagram



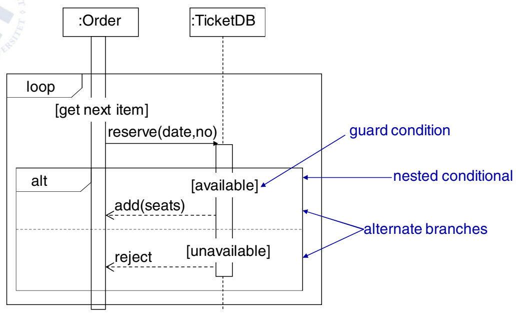
Sequence diagram with several objects



Combining fragments of interaction diagrams



More fragments of interaction diagrams



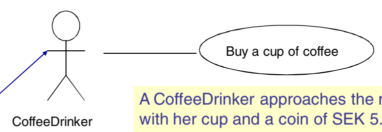
Use-case modelling

A use-case is:

"… a particular form or pattern or exemplar of usage, a scenario that begins with some user of the system initiating some transaction of sequence of interrelated events."

Jacobson et al.1992: Object-oriented software engineering. Addison-Wesley

Use-case diagram

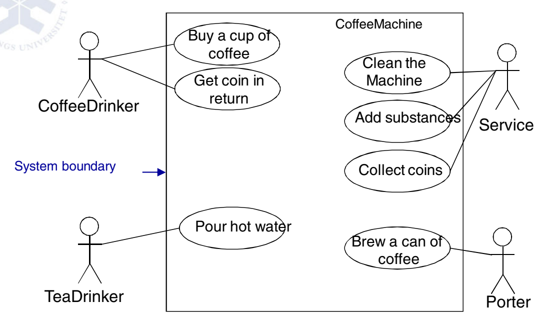


Actor: a user of the system in a particular role. Can be human or system.

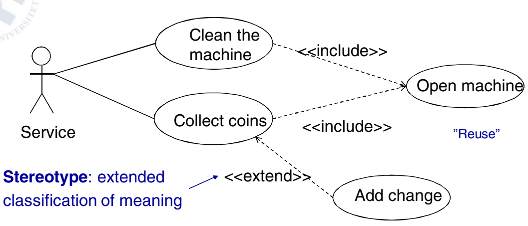
Detail of use-case

A CoffeeDrinker approaches the machine with her cup and a coin of SEK 5. She places the cup on the shelf just under the pipe. She then inserts the coin, and presses the button for coffee to get coffee according to default settings. Optionally she might use other buttons to adjust the strength and decide to add sugar and/or whitener. The machine processes the coffee and rings a bell when it is ready. The CoffeeDrinker takes her cup from the shelf.

Use-case diagram for the coffee machine



Relations between use-cases



Stereotype: extended classification of meaning

"Separating scenarious" (often conditional)

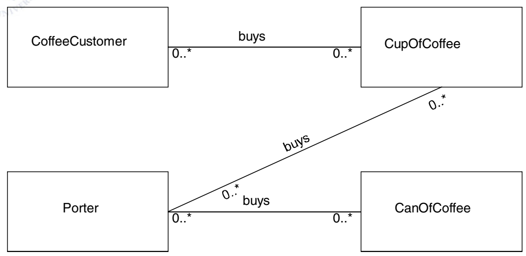
Please, keep as simple as possible.

Identifying classes: noun analysis

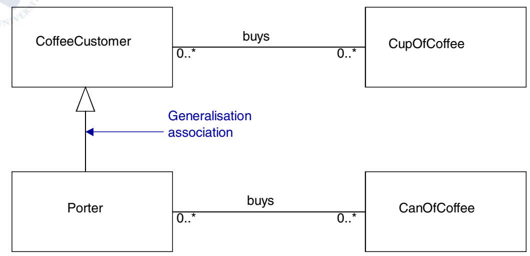
A CoffeeDrinker approaches the <u>machine</u> with her <u>cup</u> and a <u>coin</u> of SEK 5. She places the cup on the <u>shelf</u> just under the <u>pipe</u>. She then inserts the coin, and presses the <u>button</u> for coffee to get coffee according to default settings. Optionally, she might use other buttons to adjust the strength and decide to add <u>sugar</u> and/or <u>whitener</u>. The machine processes the <u>coffee</u> and rings a bell when it is ready. The CoffeeDrinker takes her cup from the shelf.

- **machine – real noun handled by the system**
- cup – unit for beverage
- coin – detail of user and machine
- shelf – detail of machine
- pipe – detail of machine
- **button– handled by the system**
- sugar – detail of coffee
- whitener – detail of coffee
- **cup of coffee – handled by the system**
- **indicator – not discovered**

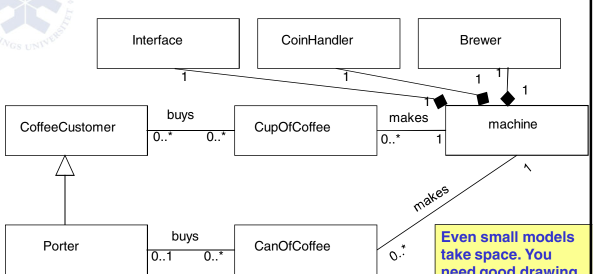
Extended class model



Revised class model

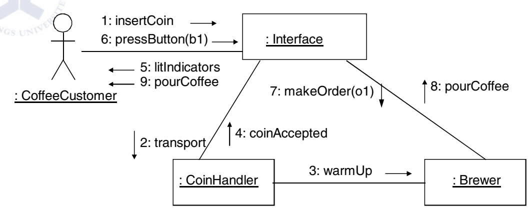


The coffee machine class model



Even small models take space. You need good drawing tools and a large sheet.

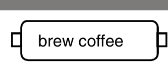
Communication diagram



Shows message flows with sequence numbers

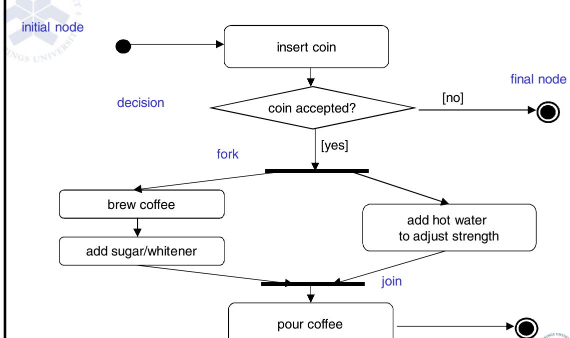
Similar information as sequence diagram

Activity Diagram



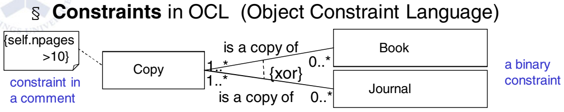
- **Graph**
  - Nodes are activities (actions)
    - Method invocations, operations, sending / receiving messages, handling events, creating / accessing / modifying / deleting objects, variables …
    - Data flow by input and output parameter pins
  - Edges are control flow transitions
  - To some degree dual to the state diagram
- Might be refined to a low-level specification;
  - cf. control flow graph (~ compiler IR)
- A Petri Net
  - Interpretation by moving tokens along edges
  - Models concurrency by multiple tokens for ”current state”
  - Fork / join for synchronization
- Models real-world workflows

Activity diagram



Other UML features…

- **Comments**
- **Constraints** in OCL (Object Constraint Language)



- **Profiles**: Collections of stereotypes for specific domains, e.g. Realtime-profile for UML
  - Customize (specialize) UML elements, e.g. associations
  - Can introduce own symbols
  - More in the lecture on Model-Driven Architecture…
- **MOF** (Meta-Object Facility):
  - UML is specified in UML MOF, a core subset of UML
  - MOF is the meta-model of UML – a language to define UML
  - Powerful mechanism for extending UML by adding new language elements

UML Summary

- UML – the standard for modeling software
- Modeling before/during design, precedes coding
- Different diagrams for different views
  - Model a software system only partially, focus on a certain aspect and/or part at a time
  - Problem: Maintaining consistency across diagrams
- UML is customizable and extendible: Profiles, MOF
- UML is semi-formal, messy, imprecise
- Tools
- Trend towards more detailed modeling
  - Stepwise refinement
  - ”executable UML”: UML 2 is almost a programming language…
- Trend towards automatized partial generation of models and code from models (MDA – model-driven architecture)

Homework Exercise

- Draw a class diagram for the following scenario:

A customer, characterized by his/her name and phone number, may purchase reservations of tickets for a performance of a show. A reservation of tickets, annotated with the reservation date, can be *either* a reservation by subscription, in which case it is characterized by a subscription series number, *or* an individual reservation. A subscription series comprehends at least 3 and at most 6 tickets; an individual reservation at most one ticket. Every ticket is part of a subscription series or an individual reservation, but not both. Customers may have many reservations, but each reservation is owned by exactly one customer. Tickets may be available or not, and one may sell or exchange them. A ticket is associated with one specific seat in a specific performance, given by date and time, of a show, which is characterized by its name. A show may have several performances.