THE UNIVERSITY OF NEWCASTLE AUSTRALIA

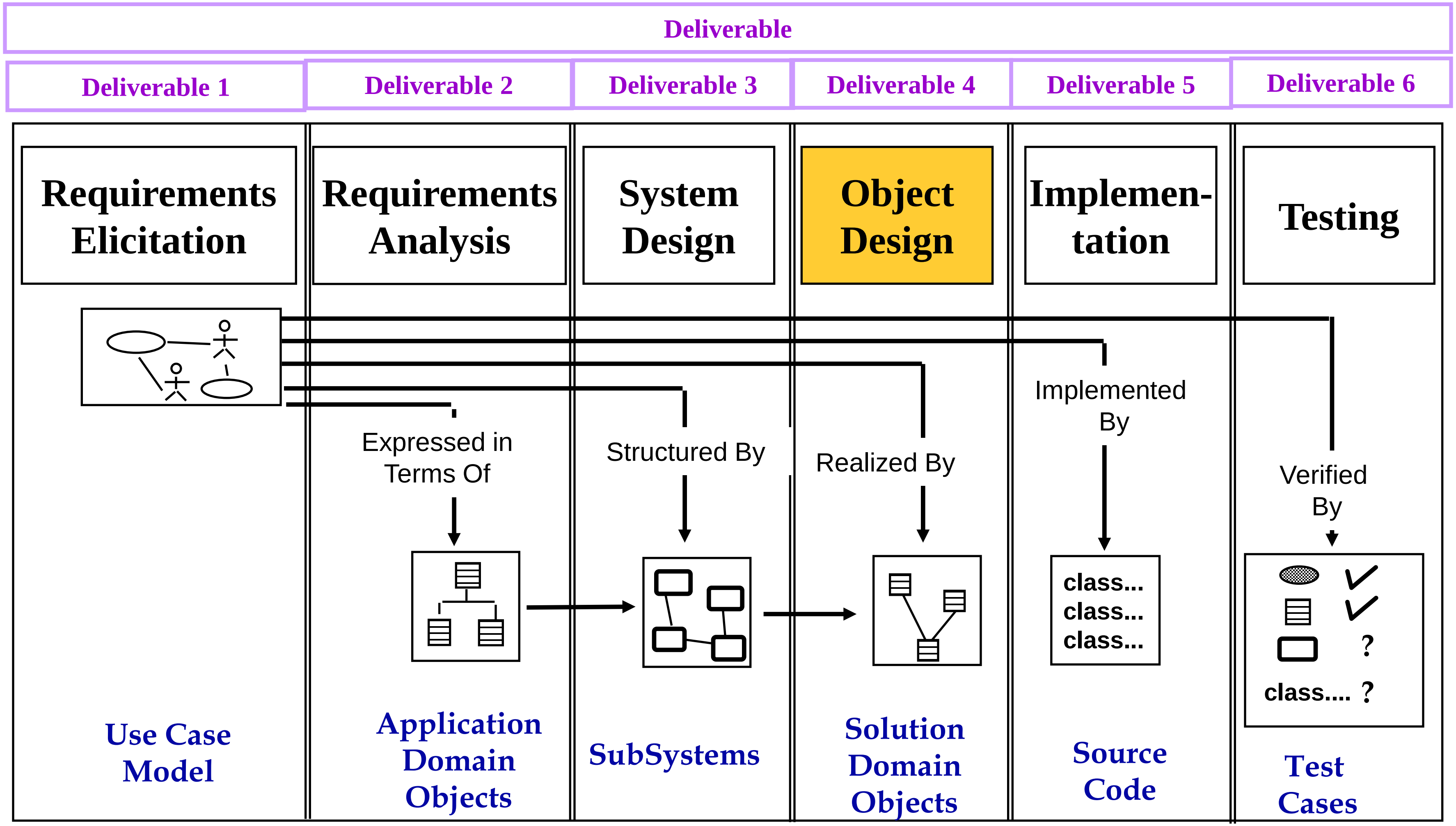
FACULTY OF ENGINEERING AND BUILT ENVIRONMENT

www.newcastle.edu.au

SENG2130 – Week 9 Object Design

SENG2130 – Systems Analysis and Design

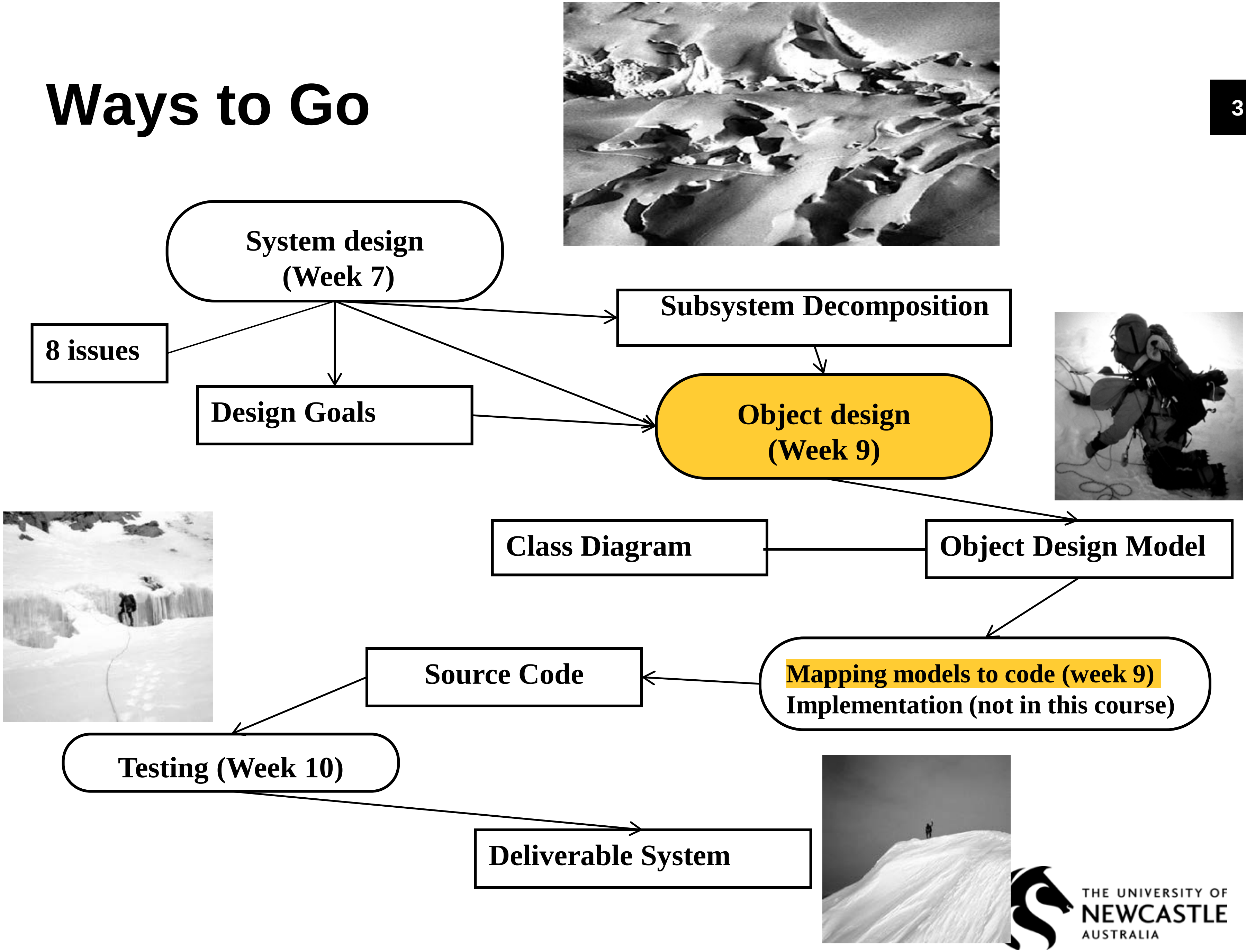
University of Newcastle



Deliverable

Deliverable 1	Deliverable 2	Deliverable 3	Deliverable 4	Deliverable 5	Deliverable 6
Requirements Elicitation	Requirements Analysis	System Design	Object Design	Implemen-tation	Testing
	Expressed in Terms Of	Structured By	Realized By	Implemented By	Verified By
Use Case Model	Application Domain Objects	SubSystems	Solution Domain Objects	Source Code	Test Cases

Ways to Go



System design (Week 7)

8 issues

Design Goals

Subsystem Decomposition

Object design (Week 9)

Object Design Model

Class Diagram

Mapping models to code (week 9)
Implementation (not in this course)

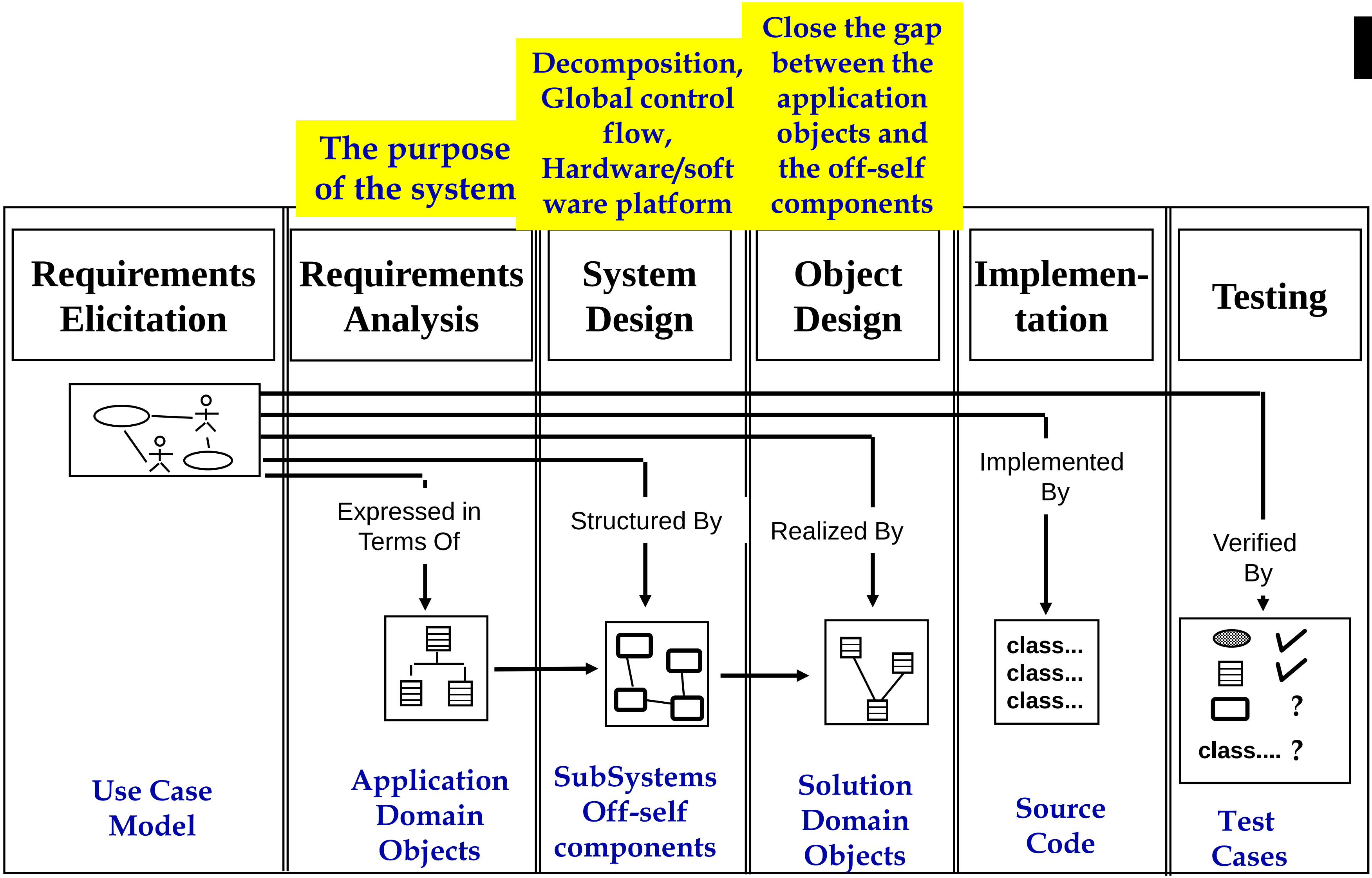
Source Code

Testing (Week 10)

Deliverable System

Object Design

- Object design is the process of adding details to the requirements analysis and making implementation decisions
  - we identify and refine solution objects to realize the subsystems defined during system design
- Requirements Analysis: Use cases, functional and dynamic model deliver operations for the object model
- Object Design: We iterate on where to put these operations in the object model
- Thus, object design serves as the basis of implementation
  - The object designer can choose among different ways to implement the system model obtained during requirement analysis



The purpose of the system

Decomposition, Global control flow, Hardware/software platform

Close the gap between the application objects and the off-self components

Requirements Elicitation

Requirements Analysis

System Design

Object Design

Implementation

Testing

Expressed in Terms Of

Structured By

Realized By

Implemented By

Verified By

Use Case Model

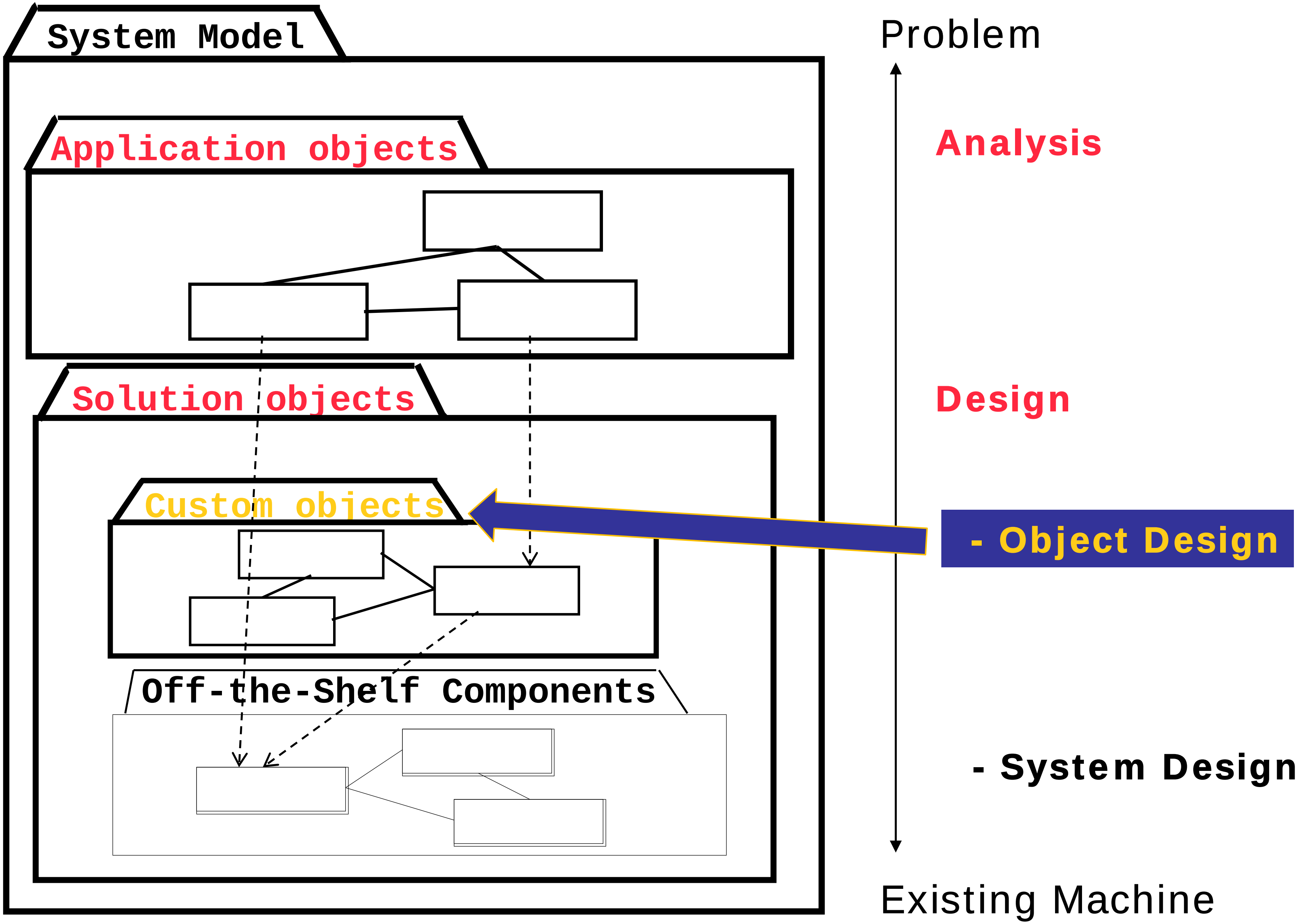
Application Domain Objects

SubSystems Off-self components

Solution Domain Objects

Source Code

Test Cases



System Model

Application objects

Solution objects

Custom objects

Off-the-Shelf Components

Problem

Analysis

Design

- Object Design

- System Design

Existing Machine

Build Custom Objects

- Problem: Close the object design gap
- How?
  1. Reuse
     - Reuse knowledge from previous experience
     - Reuse functionality already available
  2. Develop new functionality
     - Identification of new Objects during Object design
     - Composition
     - Inheritance
  3. Contract
     - Invariants
     - Preconditions
     - postconditions

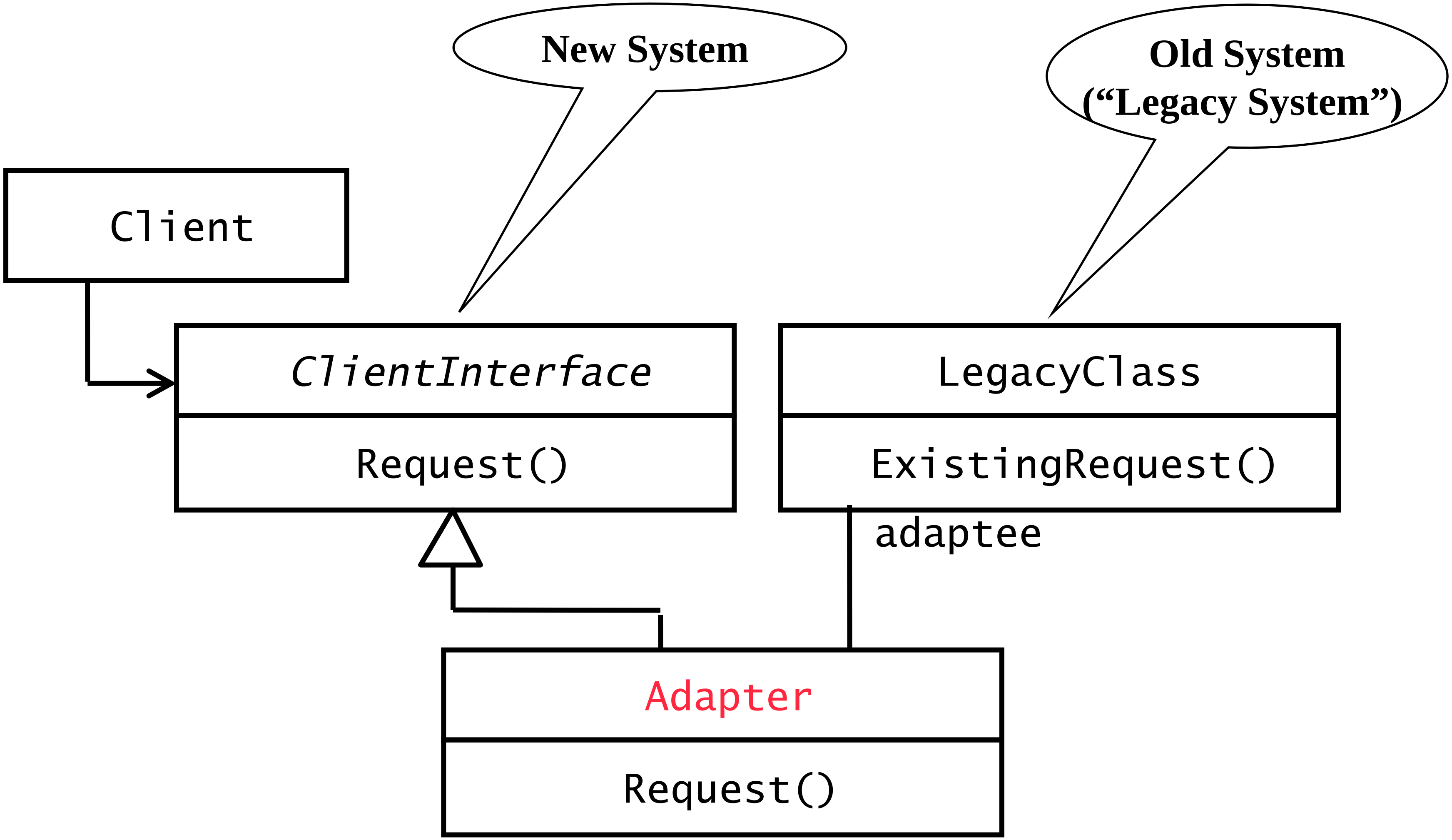
1. Reuse

- Identification of existing solutions
  - Off-the-shelf components and additional solution objects
    - Identified during system design.
    - Are used to help in the realization of each subsystem.
    - Examples: middleware, user interface toolkits, application frameworks, class libraries and class libraries of banking objects.
    - Buy-versus-Build trade-offs
  - **Object design patterns** are template solutions
    - Adapter pattern
    - Bridge pattern
    - Strategy pattern

Adapter pattern

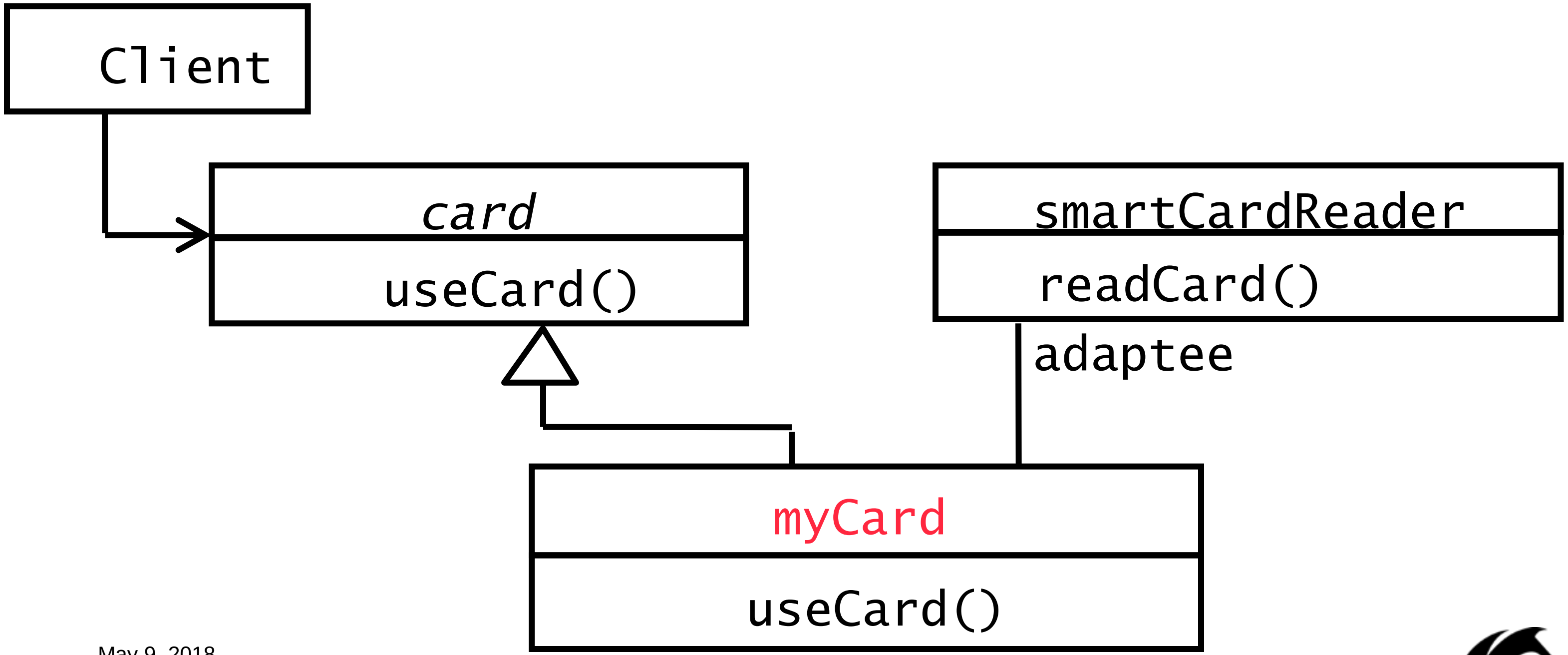
- Connects incompatible components
- Used to provide a new interface to existing legacy components

Adapter pattern (cont.1)



Adapter pattern (cont.2)

- E.g., a smart card software system should use an adapter for a smart card reader from a specific manufacturer

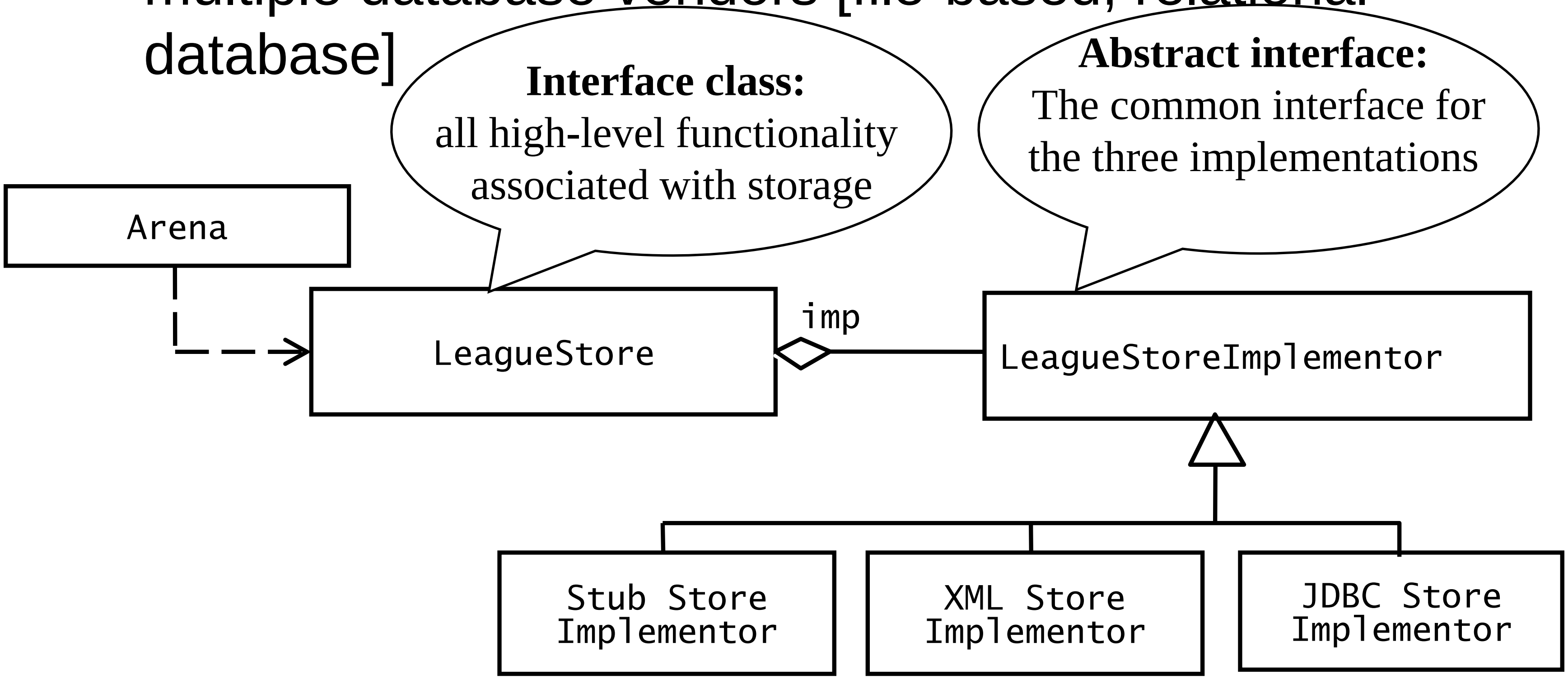


Bridge Pattern

- Use a bridge pattern to “decouple an abstraction from its implementation so that the two can vary independently”
- Use where the full set of objects is not completely known at analysis or design time
  - Use where a subsystem or component must be replaced later after the system has been deployed and client programs use it in the field
- Allows different implementations of an interface to be decided upon dynamically
- The bridge pattern can be used to provide multiple implementations under the same interface

Bridge Pattern (cont.)

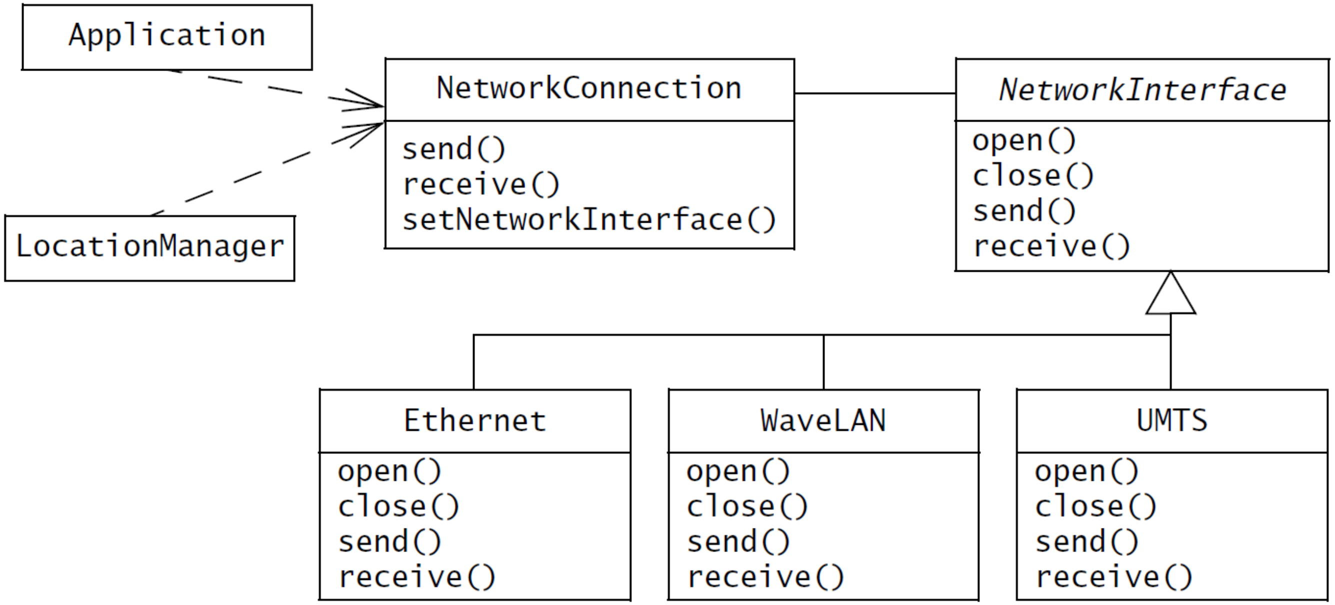
- E.g., the storage of Leagues in ARENA: support multiple database vendors [file-based, relational database]



Strategy Pattern

- Decouples an algorithm from its implementations. It encapsulates a behavior
- Eg., a mobile application running on a wearable computer
  - Uses different networks protocols depending on the location of the user
    - The shop: a local wireless network
    - On the roadside: mobile phone network
  - A car mechanic using the wearable computer to access repair manuals and maintenance records for the vehicle under repair.

Str

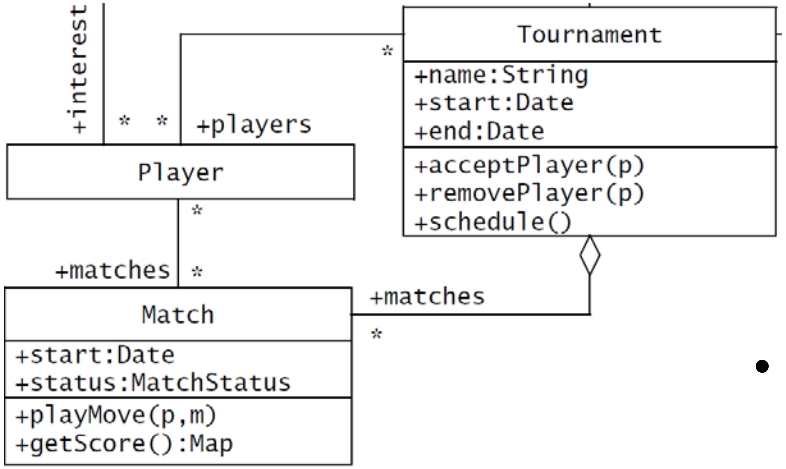


- Applying the Strategy pattern for encapsulating multiple implementations of a NetworkInterface. The LocationManager implementing a specific policy configures NetworkConnection with a concrete NetworkInterface (i.e., the mechanism) based on the current location. The Application uses the NetworkConnection independently of concrete NetworkInterfaces.

2. Develop new functionality

- Identification of new Objects during Object Design
  - Examples: ARENA
    - Analysis model: the responsibility of the TournamentStyle class is to map a list of Players participating in a Tournament onto a series of Matches.

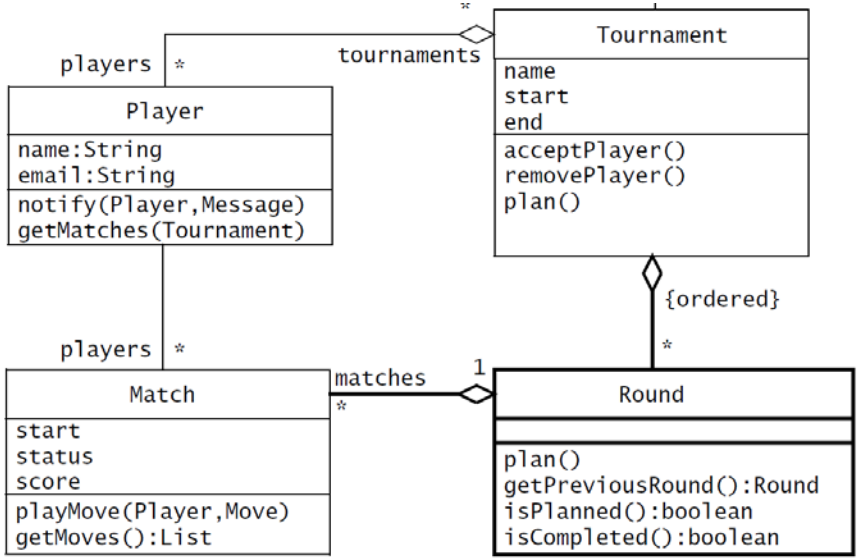
2. Develop new functionality (cont.1)



- Series of Matches that can be played in parallel (e.g., the first round of a championship) and series of Matches that have precedence constraints (e.g., both semifinals of a knock-out tournament must be completed before the final round can start).

2. Develop new functionality (cont.2)

- Identifying missing attributes and operations



- A new class Round
- A schedule for a Tournament is simply a list of Rounds
- A list of Rounds is responsible for creating all the Matches in the Tournament and organizing them into a sequence of Rounds.

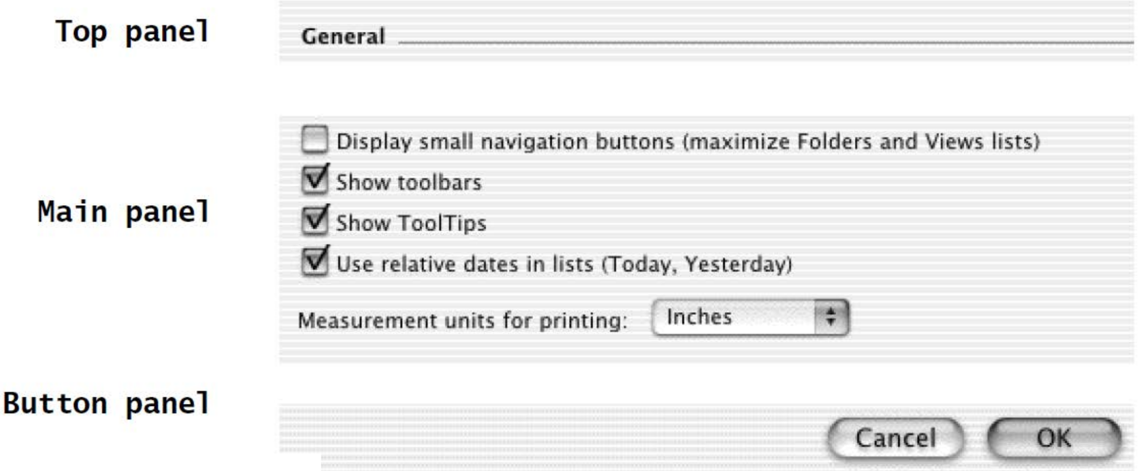
2. Develop new functionality (cont.3)

- Composition
  - new functionality is obtained by aggregation
  - The new object with more functionality is an aggregation of existing objects
- Inheritance
  - New functionality is obtained by inheritance

Composition

- Encapsulates hierarchies by providing a common superclass for aggregate and leaf nodes
- E.g., user interface toolkits (Swing, Cocoa)
  - Each class implements a specialized behavior
    - Inputting text, selecting and deselecting a check box, pushing a button, or pulling down a meu
  - The user interface design can aggregate these components into Windows

Composition



General

□ Display small navigation buttons (maximize Folders and Views lists)

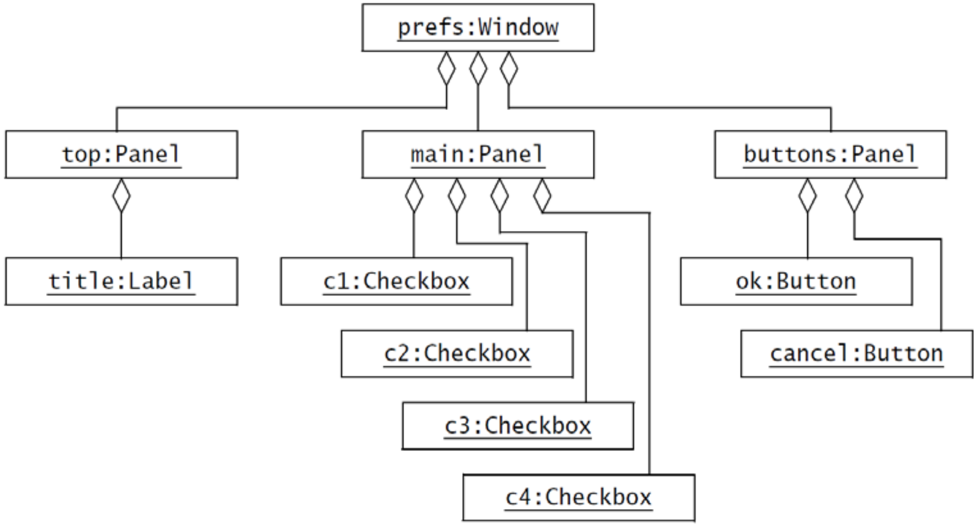
☑ Show toolbars

☑ Show ToolTips

☑ Use relative dates in lists (Today, Yesterday)

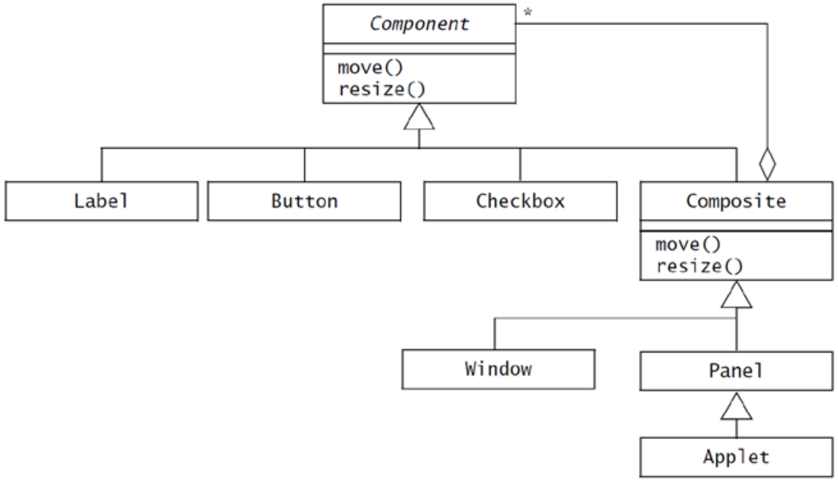
Measurement units for printing: Inches

Cancel OK



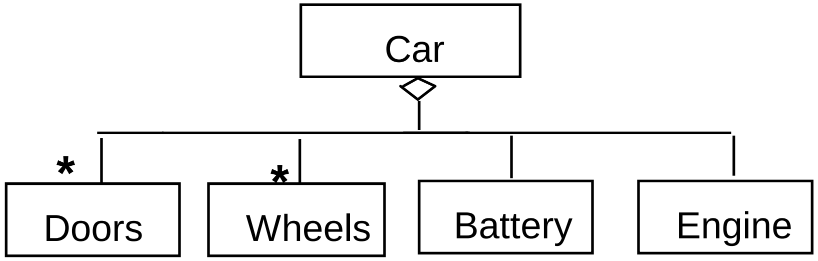
UML object diagram (aggregation)

Recursive aggregate



Composition

Fixed Structure:



Car

*

*

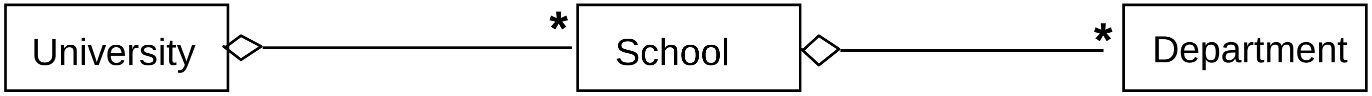
Doors

Wheels

Battery

Engine

Organization Chart (variable aggregate):



University

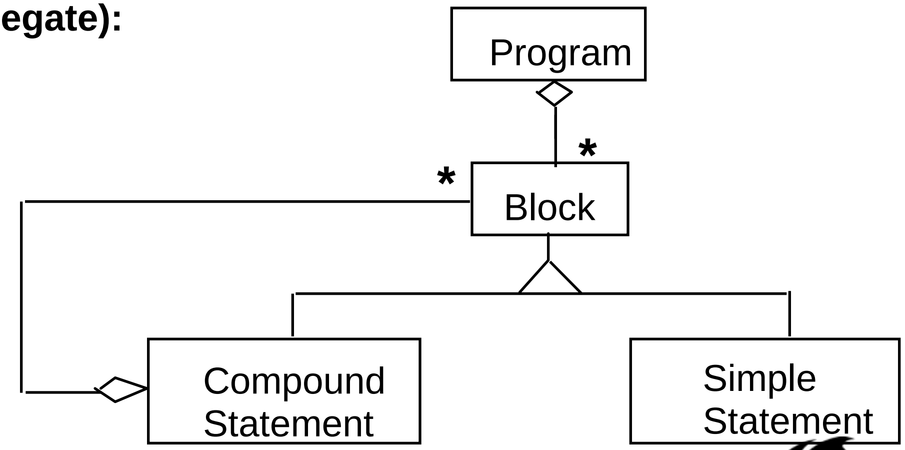
*

School

*

Department

Dynamic tree (recursive aggregate):



Program

*

*

Block

Compound Statement

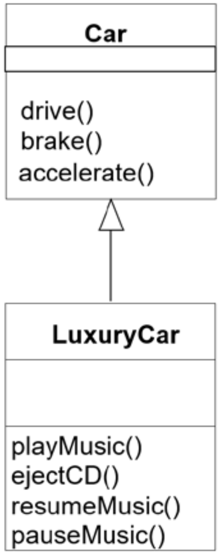
Simple Statement

Inheritance

- During analysis
  - Inheritance to classify objects into taxonomies
  - Superclass (base class) : the common behavior of the general case
  - Subclass (derived classes): the behavior that is specific to specialized objects
  - Examples: FRIEND
    - During Requirements Elicitation: focus on understanding how the system deals with `Incidents` in general, and then move to the differences in handling `Traffic Accidents` or `Fires`
    - During Object design: reduce redundancy and enhance extensibility

Inheritance (cont.1)

- Starting Point is always the requirements analysis phase:
  - We start with use cases
  - We identify existing objects (“class identification“)
  - We investigate the relationship between these objects; “Identification of associations“:
    - general associations
    - aggregations
    - inheritance associations.

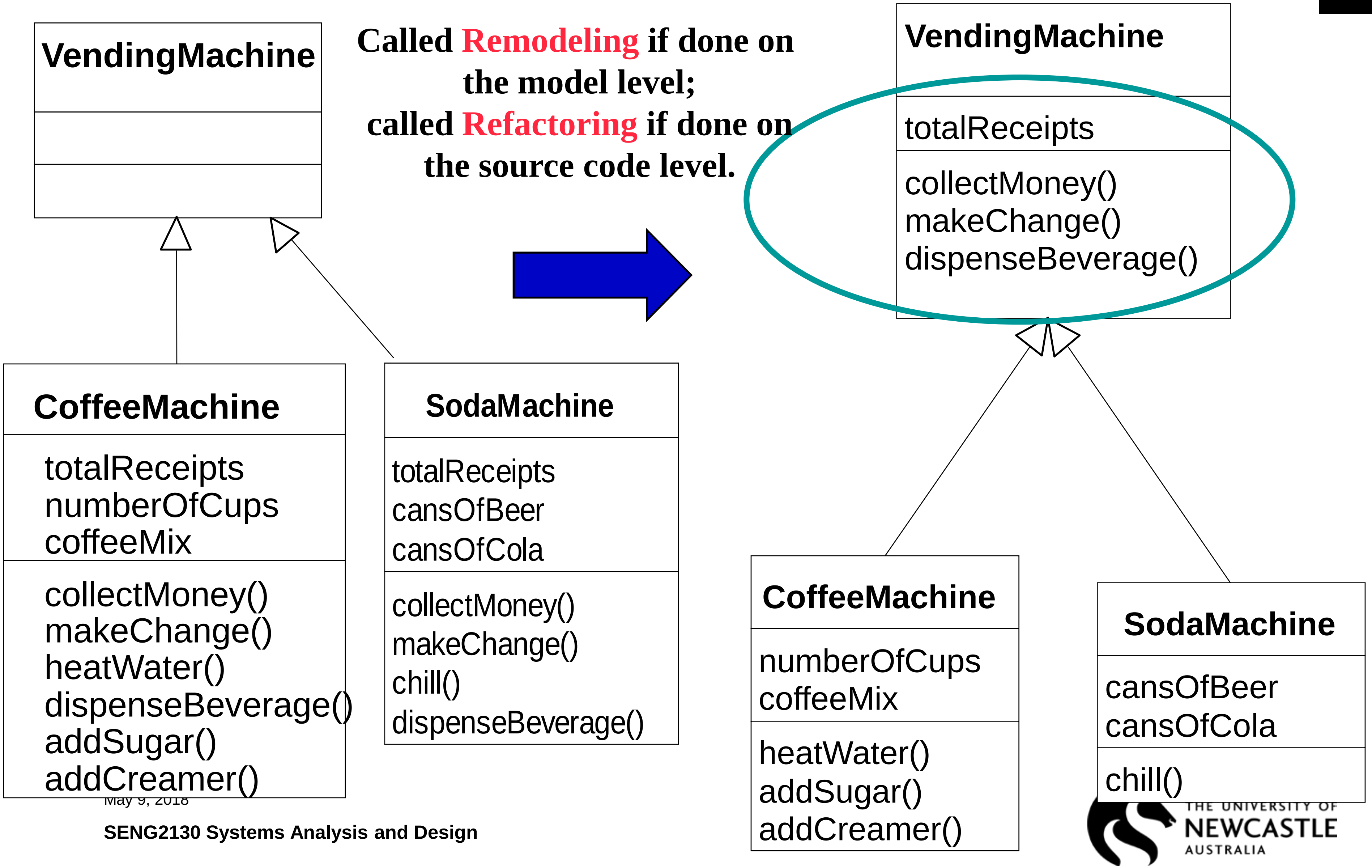


Inheritance (cont.2)

- To “discover“ inheritance associations, we can proceed in two ways, which we call specialization and generalization
- Generalization: the discovery of an inheritance relationship between two classes, where the sub class is discovered first.
  - Biology: First we find individual animals (Elephant, Lion, Tiger), then we discover that these animals have common properties (mammals)
- Specialization: the discovery of an inheritance relationship between two classes, where the super class is discovered first.

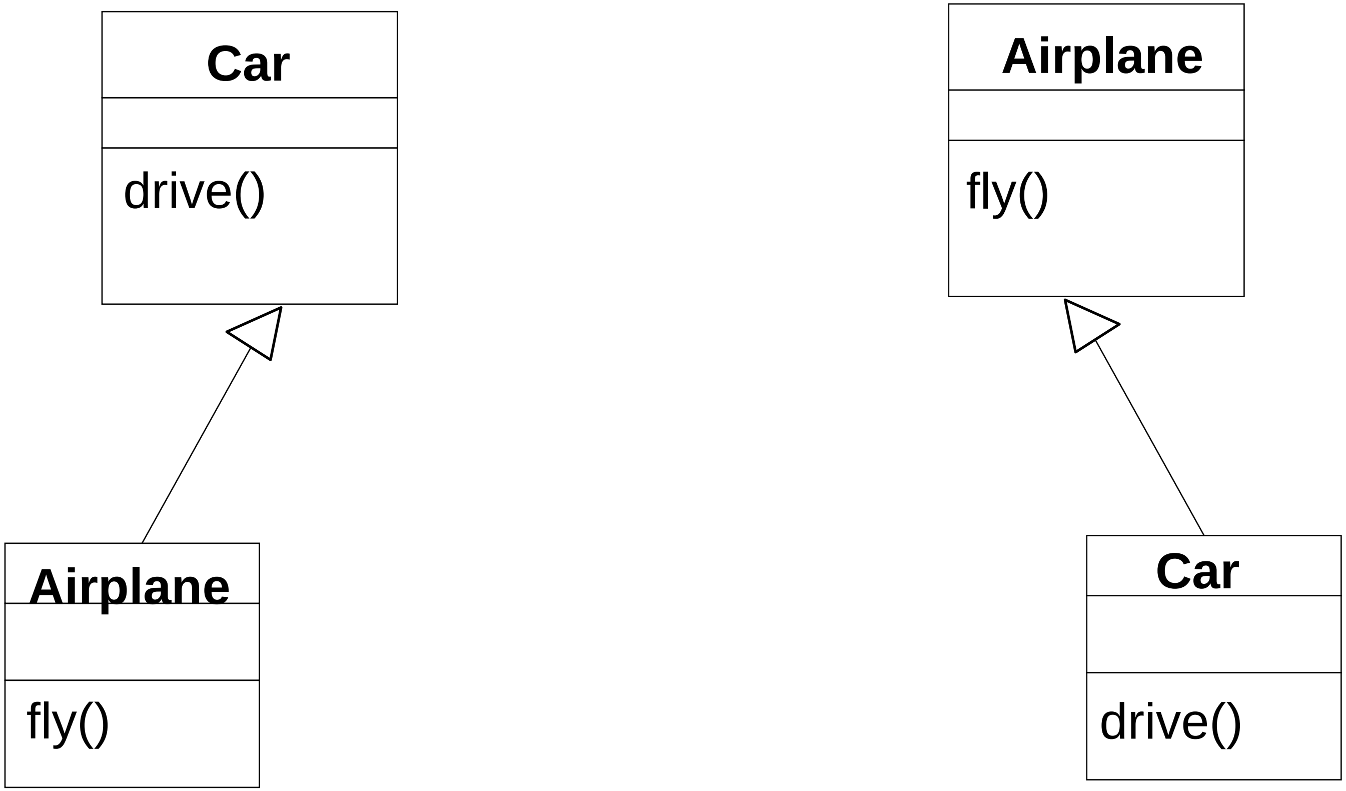
Restructuring of Attributes and Operations is often a Consequence of Generalization

Called **Remodeling** if done on the model level; called **Refactoring** if done on the source code level.

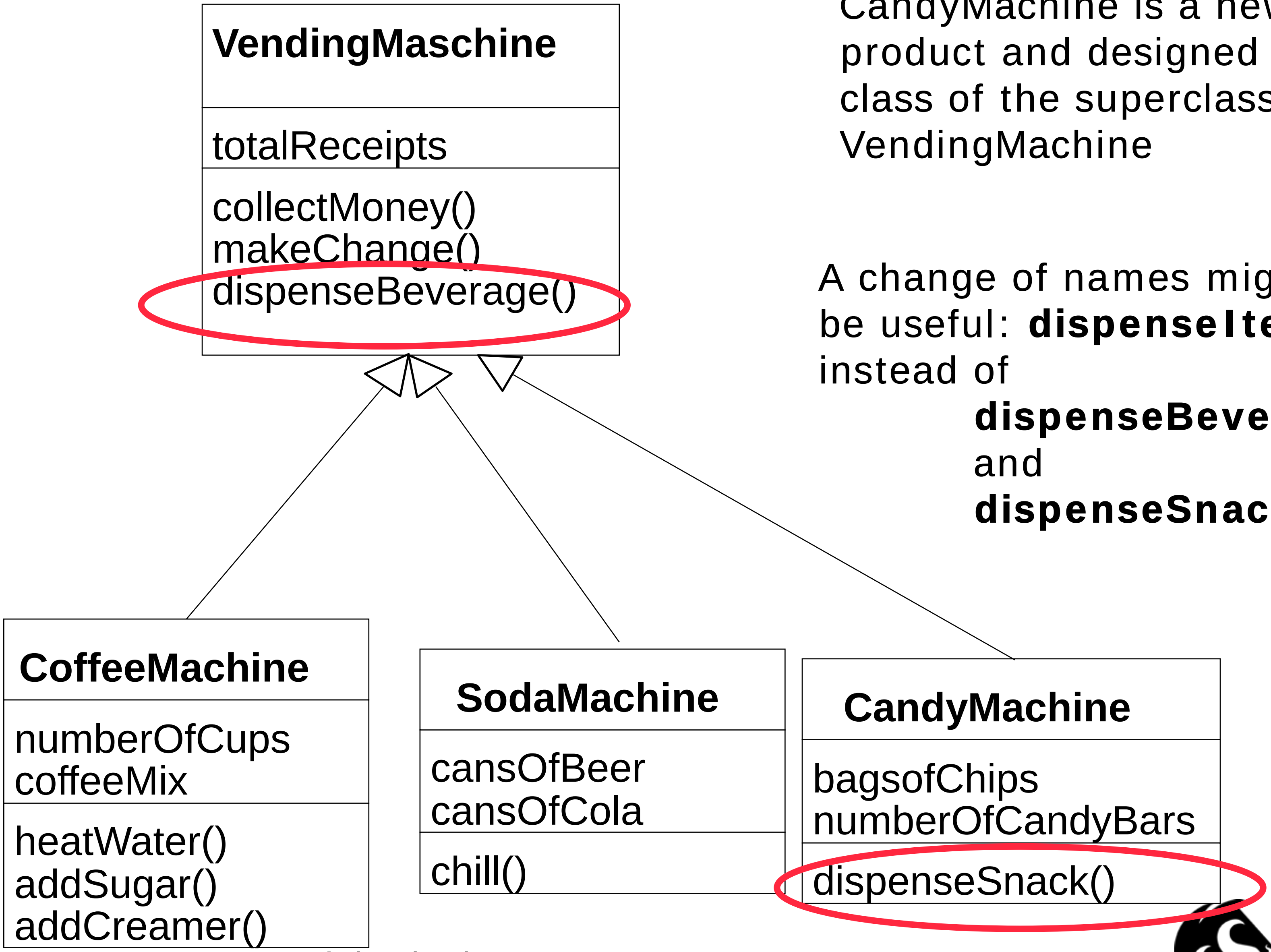


Inheritance (cont.3)

- Which taxonomy is correct ?



Specialization example



VendingMaschine

totalReceipts

collectMoney()
makeChange()
dispenseBeverage()

CoffeeMachine

numberOfCups
coffeeMix

heatWater()
addSugar()
addCreamer()

SodaMachine

cansOfBeer
cansOfCola

chill()

CandyMachine

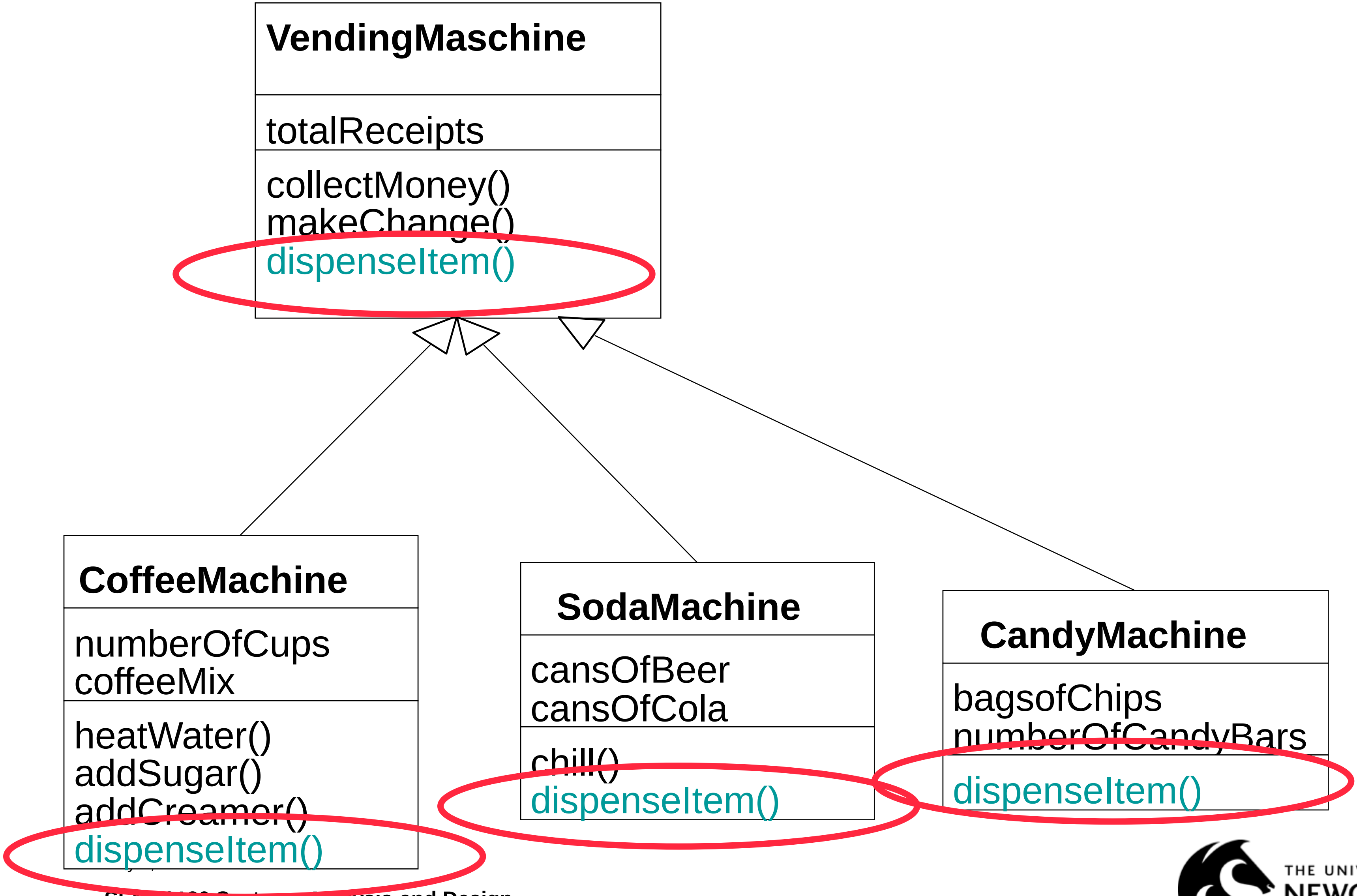
bagsofChips
numberOfCandyBars

dispenseSnack()

CandyMachine is a new product and designed as a sub class of the superclass VendingMachine

A change of names might now be useful: **dispenseItem()** instead of **dispenseBeverage()** and **dispenseSnack()**

Specialization example



VendingMaschine

totalReceipts

collectMoney()
makeChange()
dispenseItem()

CoffeeMachine

numberOfCups
coffeeMix

heatWater()
addSugar()
addCreamer()
dispenseItem()

SodaMachine

cansOfBeer
cansOfCola

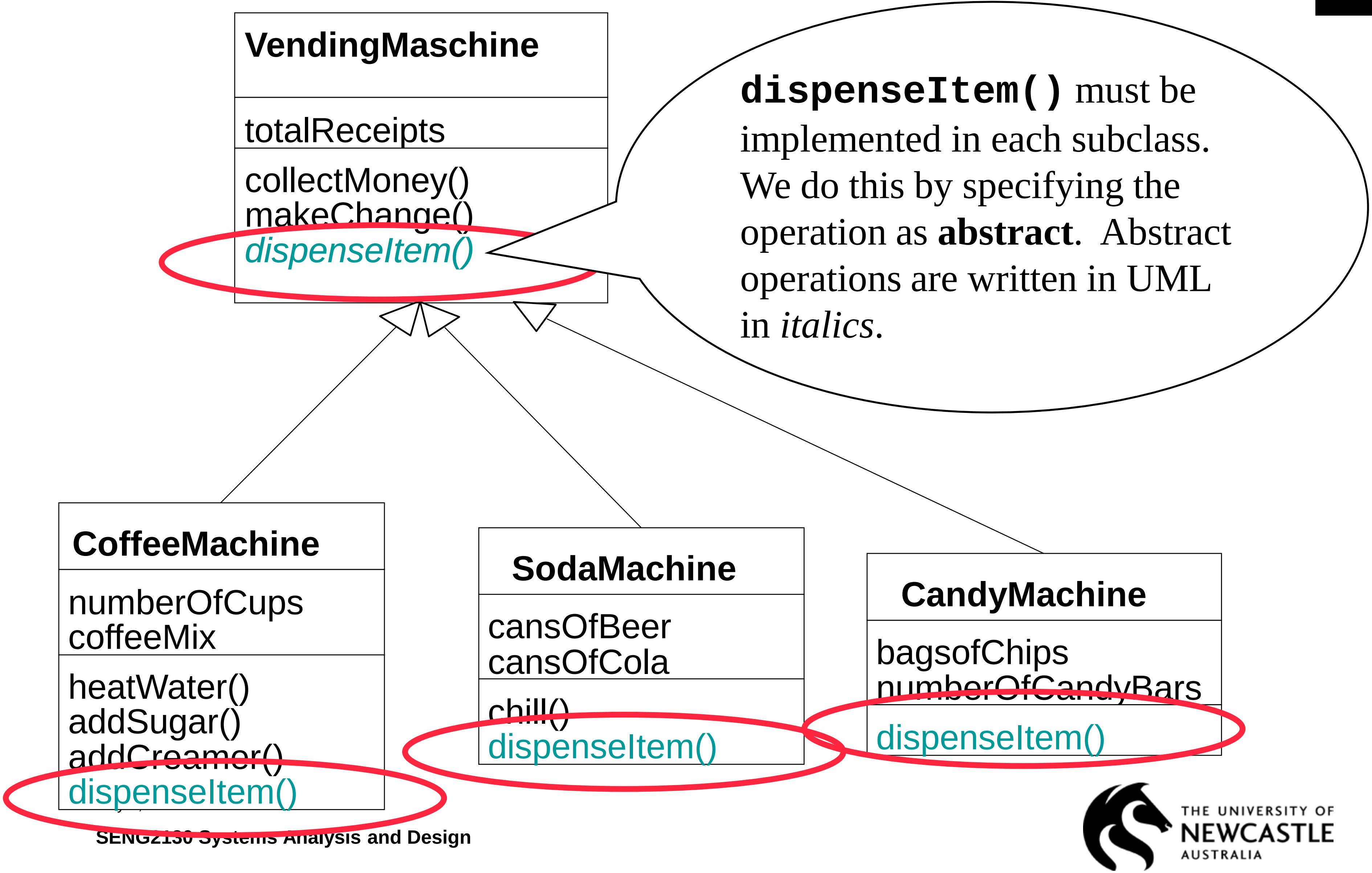
chill()
dispenseItem()

CandyMachine

bagsofChips
numberOfCandyBars

dispenseItem()

Specialization example – Abstract Method



VendingMaschine

totalReceipts

collectMoney()
makeChange()
dispenseItem()

`dispenseItem()` must be implemented in each subclass. We do this by specifying the operation as **abstract**. Abstract operations are written in UML in *italics*.

CoffeeMachine

numberOfCups
coffeeMix

heatWater()
addSugar()
addCreamer()
dispenseItem()

SodaMachine

cansOfBeer
cansOfCola

chill()
dispenseItem()

CandyMachine

bagsofChips
numberOfCandyBars

dispenseItem()

3. Contracts

- Constrains on a class that enable class users, implementers and extenders
  - Invariant: it is used to specify consistency constraints among class attributes
  - Precondition: it used to specify constraints that a class user must meet before calling the operation
  - Postcondition: it used to specify constraints that a class implementer and the class extender must ensure after the invocation of the operation

3. Contracts (cont.)

- Examples:

  acceptPlayer(): to add a **Player** in the Tournament

  removePlayer(): to withdraw a **Player** from the Tournament

  getMaxNumPlayers(): to get the maximum number of **Players** who can participate in this Tournament

  - Invariant: the maximum number of **Players** in the `Tournament` should be positive. If a Tournament is created with a `maxNmPlayers` that is `zero`, the *acceptPlayer()* method will always violate its contract and the `Tournament` will never start.

3. Contracts (cont.)

- Examples:

  acceptPlayer(): to add a **Player** in the Tournament

  removePlayer(): to withdraw a **Player** from the Tournament

  getMaxNumPlayers(): to get the maximum number of **Players** who can participate in this Tournament

  – Precondition (for *acceptPlayer()*): the **Player** to be added has not yet already been accepted and the `Tournament` has not yet reached its maximum number of **Players**

3. Contracts (cont.)

- Examples:

  acceptPlayer(): to add a **Player** in the Tournament

  removePlayer(): to withdraw a **Player** from the Tournament

  getMaxNumPlayers(): to get the maximum number of **Players** who can participate in this Tournament

  - Postcondition (for *acceptPlayer()*): the current number of **Players** must be exactly one more than the number of **Players** before the invocation of *acceptPlayer()*.

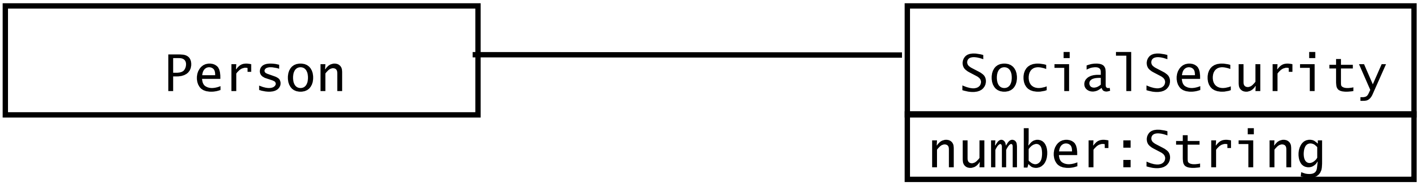
Mapping Models to Code

2 different types of transformations

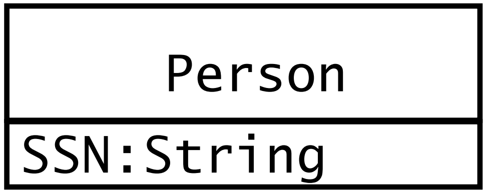
1. Model transformation
2. Forward Engineering

1. Model transformations

- Goal: optimizing the object design model
- Collapsing Objects
  - Before



  - After



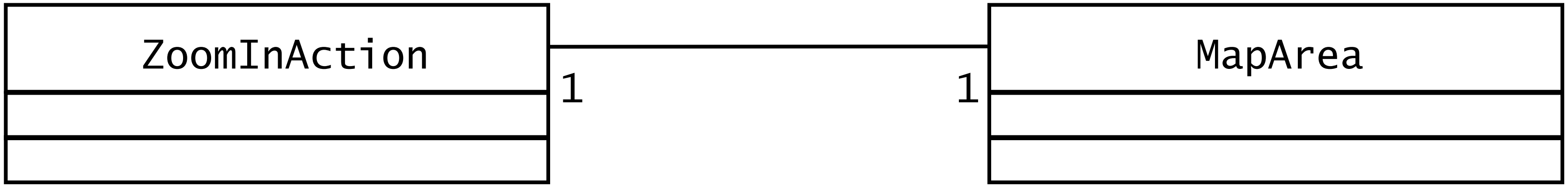
  - Turning an object into an attribute of another object is usually done, if the object does not have any interesting dynamic behavior (only get and set operations)

2. Forward Engineering – mapping inheritance

- Goal: We have a UML-Model with inheritance. We want to translate it into source code
- Program languages offer several techniques to realize the different types of inheritance
  - E.g., Java – Overwriting o methods, Final classes, Final methods, Abstract methods, Abstract classes, Interfaces

2. Forward Engineering – mapping associations (cont.1)

- Unidirectional 1-to-1
  - Before



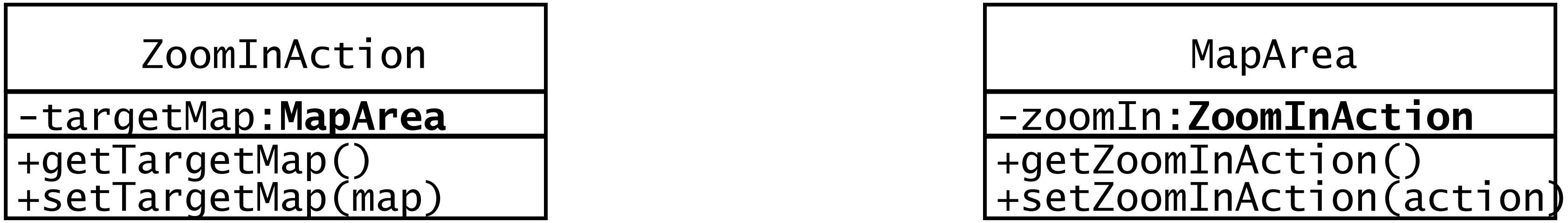
ZoomInAction

1

1

MapArea

  - After



ZoomInAction

-targetMap:**MapArea**

+getTargetMap()
+setTargetMap(map)

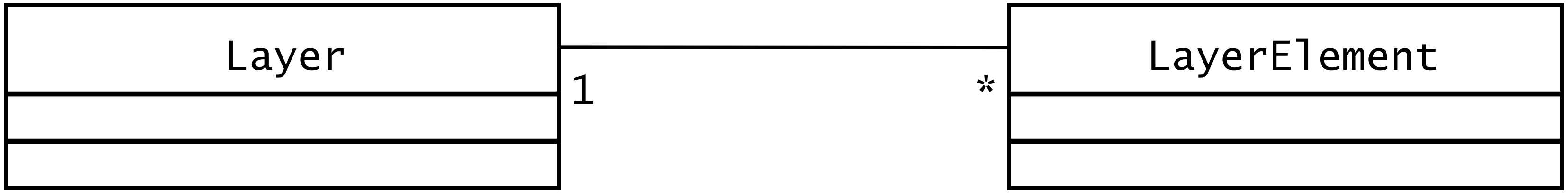
MapArea

-zoomIn:**ZoomInAction**

+getZoomInAction()
+setZoomInAction(action)

2. Forward Engineering – mapping associations (cont.2)

- 1-to-Many Association
  - Before



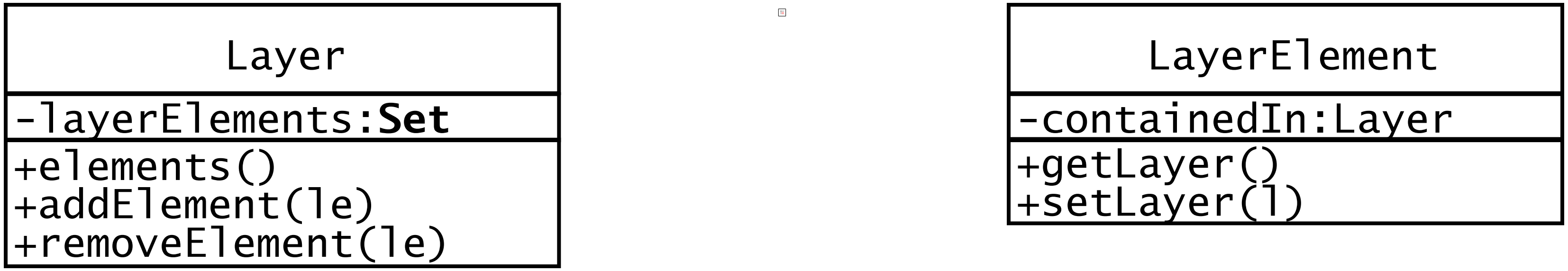
Layer

1

*

LayerElement

  - After



Layer

-layerElements:**Set**

+elements()
+addElement(le)
+removeElement(le)

LayerElement

-containedIn:Layer

+getLayer()
+setLayer(l)

2. Forward Engineering – mapping contracts to exceptions

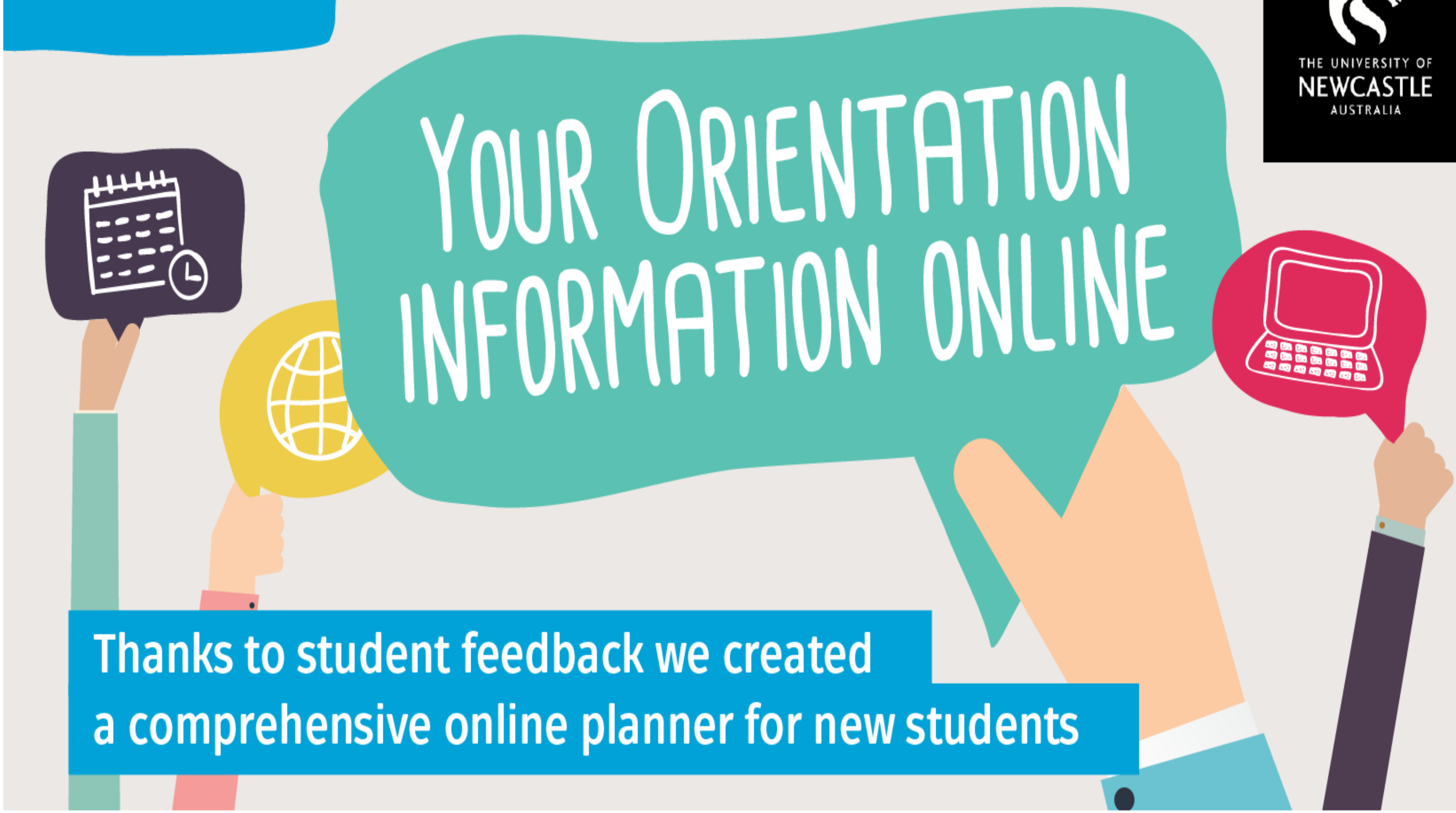
- Many object-oriented languages support exceptions
- We can use their exception mechanisms for signaling and handling contract violations
- E.g., Java : try-throw-catch mechanism

Summary

- Object design
  - Reuse (Adapt pattern, Bridge pattern, Strategy pattern)
  - Develop new objects
  - Contract
- Mapping concepts
  - Model transformation
    - Collapsing object
  - Forward engineering
    - Mapping inheritance
    - Mapping associations to collections
    - Mapping contracts to exceptions

You wanted...

Your Orientation Information Online



Thanks to student feedback we created a comprehensive online planner for new students

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Have your say:

Student Feedback on UON (SFUN) now open

Check your NUmail or log in to myUON and click ‘SURVEYS’

100% CONFIDENTIAL

You wanted...

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Thanks to student feedback we launched a new and improved MyUON app you can tailor to your needs

Have your say:

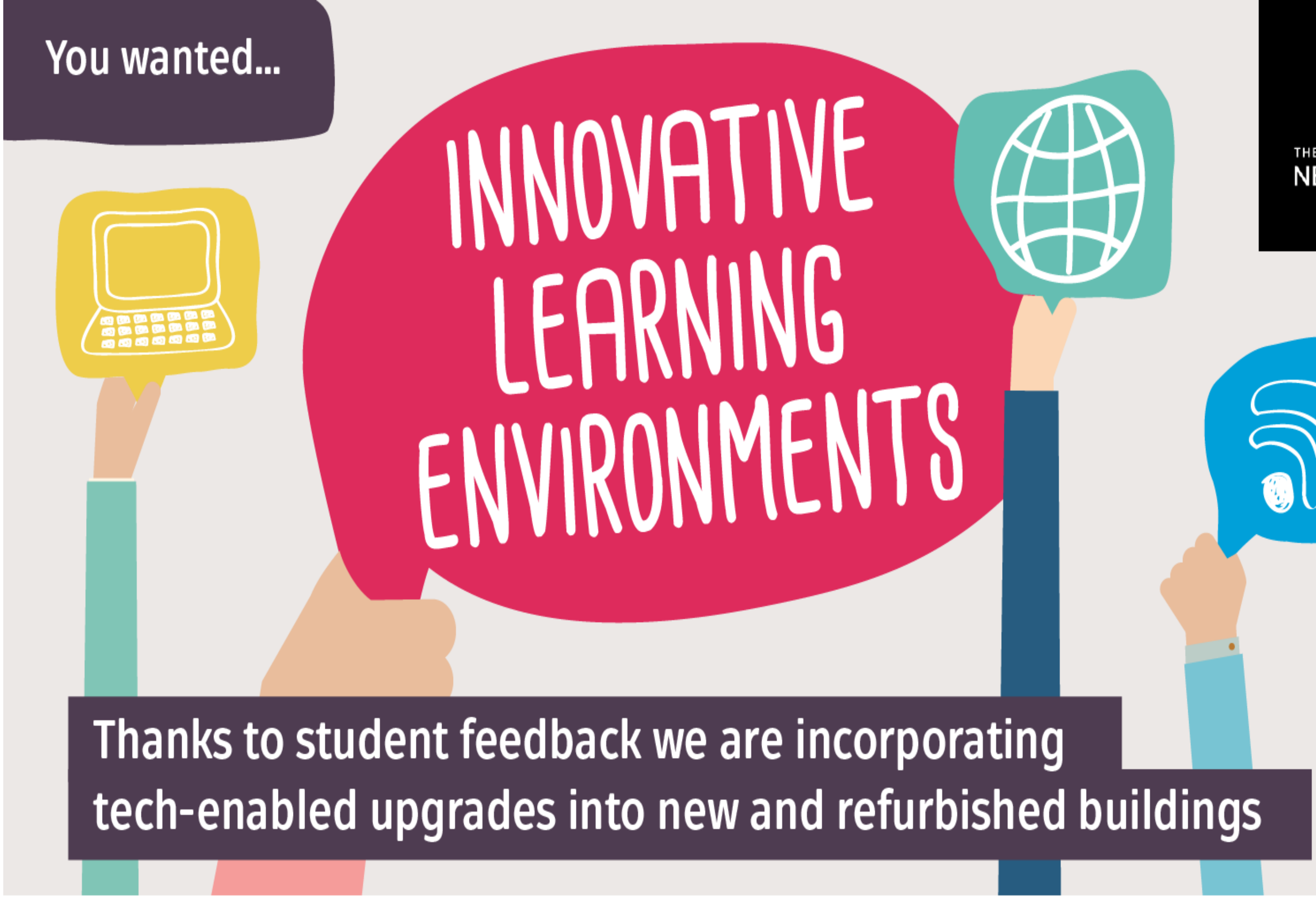
Student Feedback on UON (SFUN) now open

Check your NUmail or log in to myUON and click 'SURVEYS'

100% CONFIDENTIAL

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

You wanted...



Thanks to student feedback we are incorporating tech-enabled upgrades into new and refurbished buildings

Have your say:

Student Feedback on UON (SFUN) now open

Check your NUmail or log in to myUON and click ‘SURVEYS’

🔒 **100% CONFIDENTIAL**