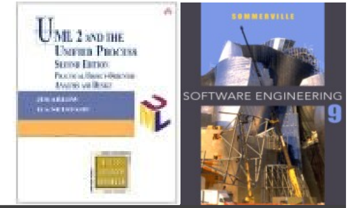
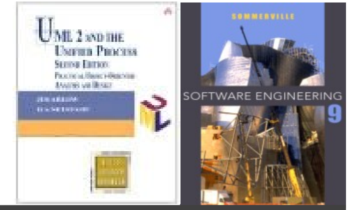


Analysis and Design

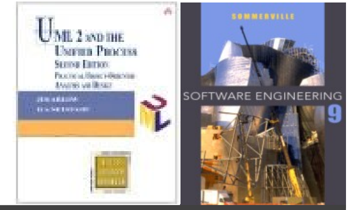
Lecture 4

Outline



- Software analysis and design
- Structured vs. object-oriented methods

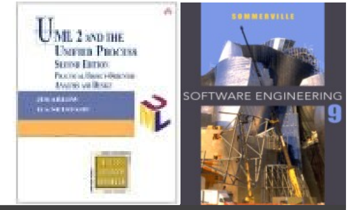
- Object-oriented analysis
- Objects and classes
- Finding analysis classes



Software Analysis and Design

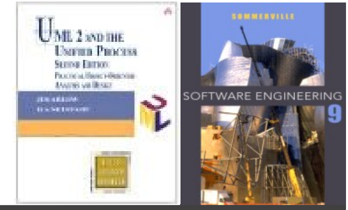
Lecture 4/Part 1

Analysis, design and implementation



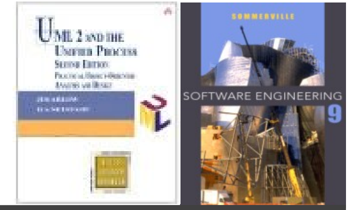
- Software development (i.e. analysis, design and implementation) is the stage in the software engineering process at which an executable software system is developed.
- Software analysis, design and implementation are invariably inter-leaved with blurred border in between.
  - **Software analysis** is a creative activity in which you identify software processes, entities (objects) and their relationships.
  - **Software design** refines analytical models with implementation details.
  - **Implementation** is the process of realizing the design as a program.

Build or buy



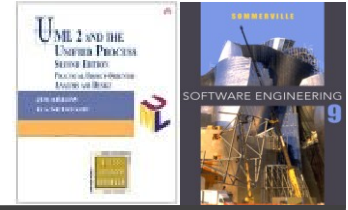
- In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
  - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

Process stages



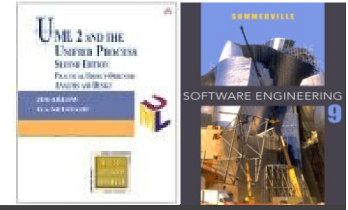
- There are a variety of different design processes that depend on the organization using the process.
- Common activities in these processes include:
  - Define the context and modes of use of the system;
  - Draft the system architecture;
  - Identify the principal system processes and entities;
  - Develop design models;
  - Specify component/object interfaces;
  - Finalize system architecture.
- Process illustrated here using a design for a wilderness weather station.

System context and interactions



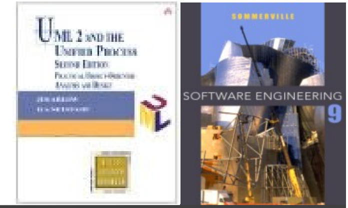
- Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

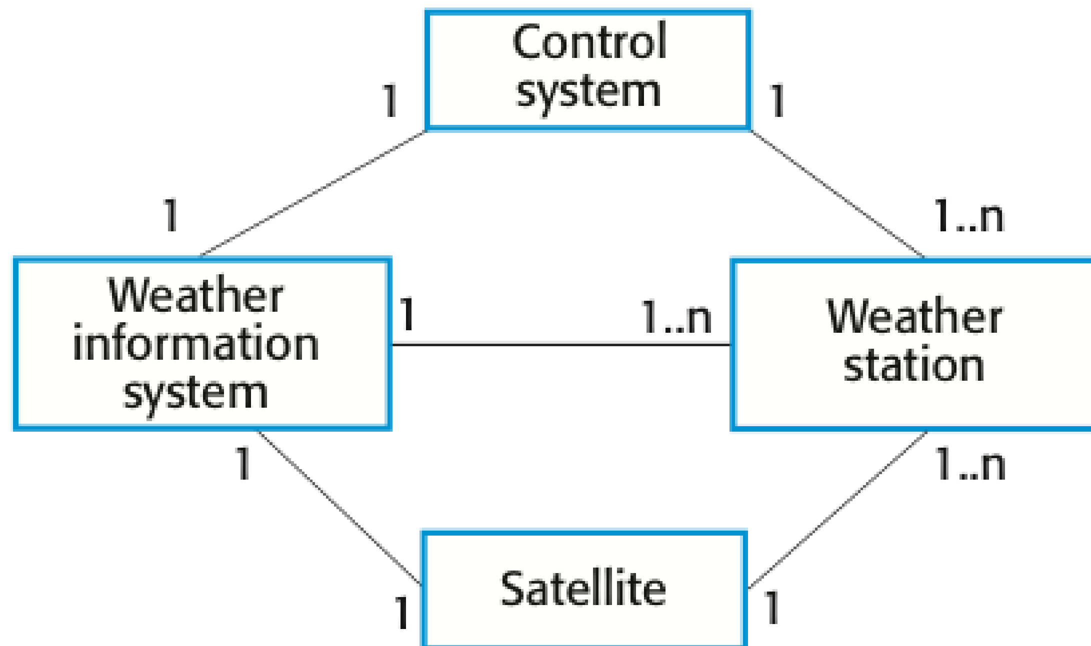
Context and interaction models



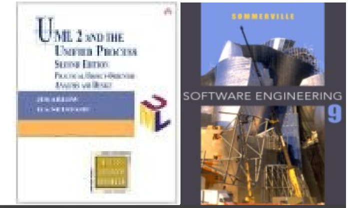
- A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
- An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

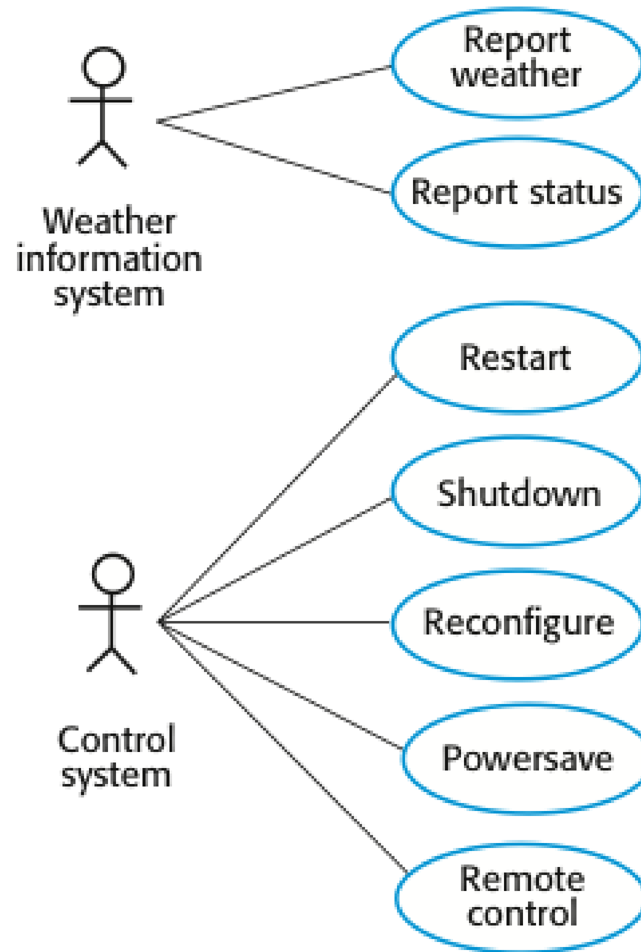
System context for the weather station



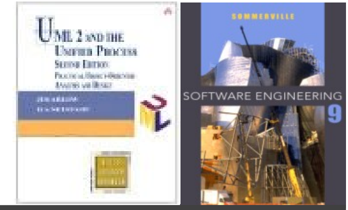


Weather station use cases



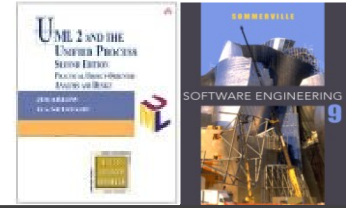


Architectural design



- May start system analysis or finish system design, often both.
- Represents the link between requirements specification and analysis/design processes.
- Often carried out in parallel with specification activities.
- It involves identifying major system components and their communications.
- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.

High-level architecture of the weather station



«subsystem»
Fault manager

«subsystem»
Configuration manager

«subsystem»
Power manager

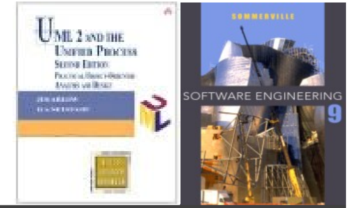
Communication link

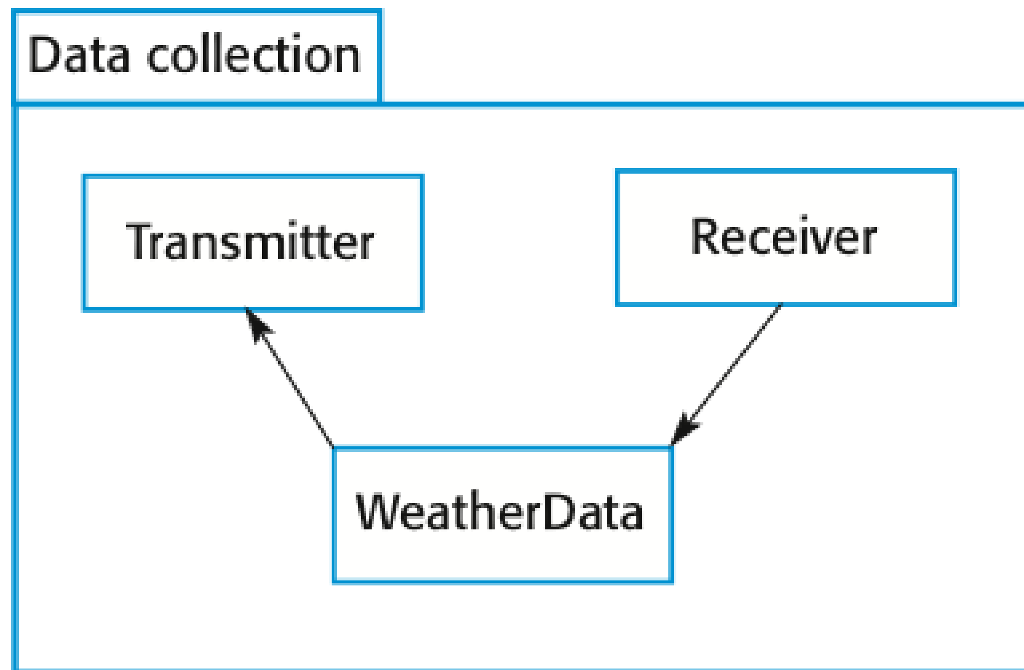
«subsystem»
Communications

«subsystem»
Data collection

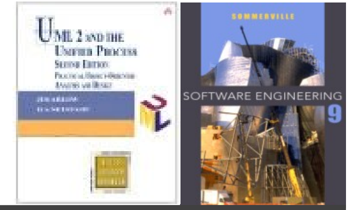
«subsystem»
Instruments

Architecture of data collection system



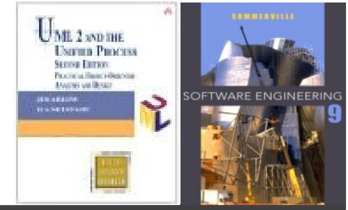


Architectural abstraction



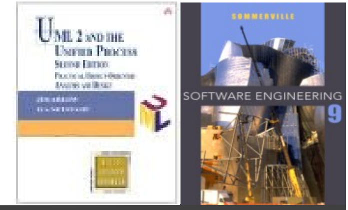
- **Architecture in the small (analysis)** is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- **Architecture in the large (design)** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Advantages of explicit architecture



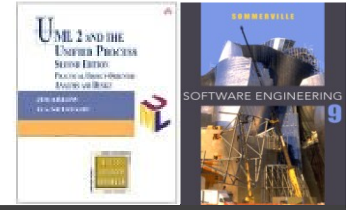
- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Product-line architectures may be developed.

Use of architectural models



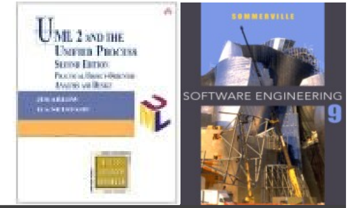
- As a way of facilitating discussion about the system analysis and design
  - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- As a way of documenting an architecture that has been designed
  - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

System analysis



- Identification of system entities (object classes in object-oriented analysis) playing the key roles in the system's problem domain, and their relationships.
- Distillation and documentation of key system processes.
- System analysis is a difficult creative activity. There is no 'magic formula' for good analysis. It relies on the skill, experience and domain knowledge of system analysts.
- Object/relationships/processes identification is an iterative process. You are unlikely to get it right first time.

Weather station object classes



WeatherStation
identifier
reportWeather ( ) reportStatus ( ) powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)

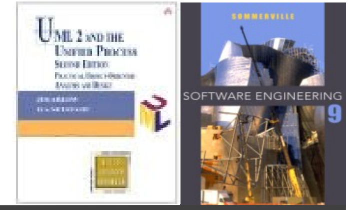
WeatherData
airTemperatures groundTemperatures windSpeeds windDirections pressures rainfall
collect ( ) summarize ( )

Ground thermometer
gt_Ident temperature
get ( ) test ( )

Anemometer
an_Ident windSpeed windDirection
get ( ) test ( )

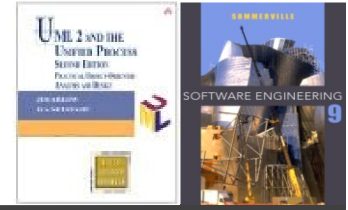
Barometer
bar_Ident pressure height
get ( ) test ( )

Design models



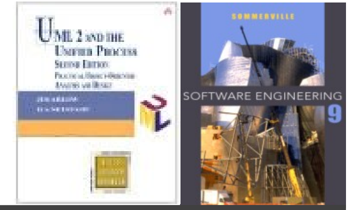
- Design models refine analysis models with the information required to communicate and document the intended implementation of the system.
- Static models describe the static structure of the system in terms of system entities and relationships.
- Dynamic models describe the dynamic interactions between entities.

Examples of design models

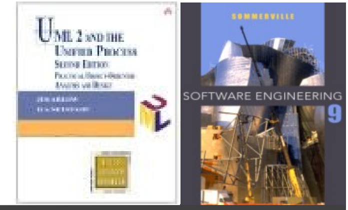


- Subsystem models that show logical groupings of objects into coherent subsystems and interface design.
- Sequence models that show the sequence of object interactions.
- State machine models that show how individual objects change their state in response to events.
- Other models include use-case models, aggregation models, generalisation models, etc.

Key points



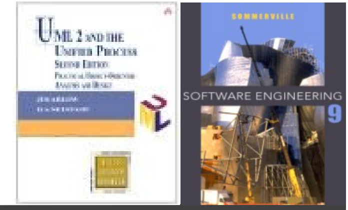
- Software analysis and design are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- The process of analysis and design includes activities to design the system architecture, identify entities in the system, describe the design using different models and document the component interfaces.
- Software analysis is a creative activity in which you identify software processes, entities (objects) and their relationships.
- Software design refines analytical models with implementation details.



Structured vs. Object-Oriented Methods

Lecture 4/Part 2

Fundamental views of software systems



✧ Function oriented view

- System as a set of interacting functions. Functional transformations based in processes, interconnected with data and control flows.

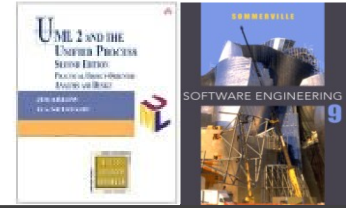
✧ Data oriented view

- Searches for fundamental data structures in the system. Functional aspect of the system (i.e. data transformation) is less significant.

✧ Object oriented view

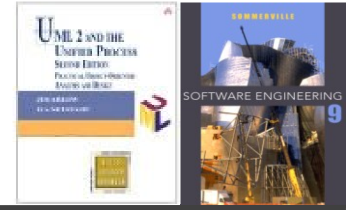
- System as a set of interacting objects, encapsulating both the data and operations performed on the data.

Structured vs. object-oriented analysis



- Structured analysis
  - Driven by the function oriented view, in synergy with data oriented view, through the concept of functional decomposition.
- Object-oriented analysis
  - Driven by the object oriented view.

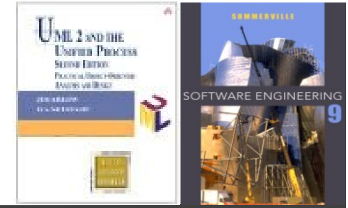
Structured analysis and design



- Divides a project on small, well defined activities and defines the order and interaction of the activities.
- Using hierarchical graphical techniques, resulting in a detailed structured specification, which can be understood by both system engineers and users.
- Effective in project structuring to smaller parts, which simplifies time and effort estimates, deliverables control and project management as such.

- Aimed at increasing system quality.

Functional decomposition



System context

Level 0 processes

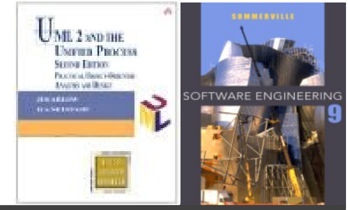
Level 1 processes

Basic-process specification

Data dictionary

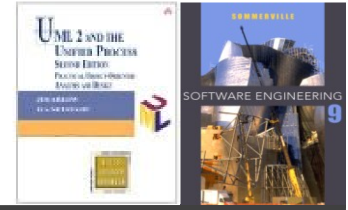
© Strukturovaná analýza systémů
by J. Ráček

Structured methods



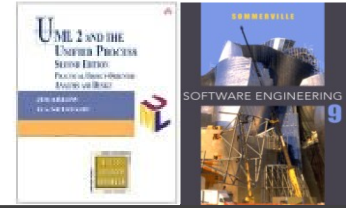
- ✧ DeMarco: Structured Analysis and System Specification (SASS)
- ✧ Gane-Sarson: Logical Modelling (LM)
- ✧ **Yourdon: Modern Structured Analysis (YMSA)**
  - Concentrates on the data and control flow of system processes and sub-processes.
- ✧ **Structured Systems Analysis and Design Method (SSADM)**
  - Physical design, logical process design and logical data design

Core notations of structured methods



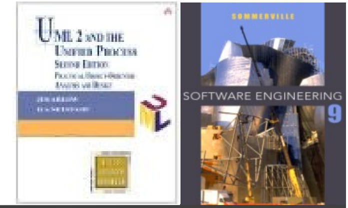
- ✧ Context diagram
  - Models system boundary and environment.
- ✧ Data flow diagram (DFD)
  - Models the system as a network of processes completing designated functions and accessing system data.
- ✧ Entity relationship diagram (ERD)
  - Models system's data.
- ✧ State diagram (STD)
  - Models system states and actions guarding transitions from one state to another.

Examplary method (Gane-Sarson)



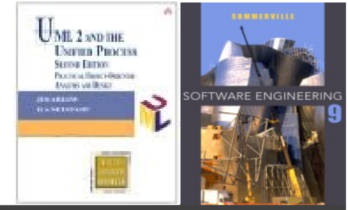
1. Define system context and create initial system DFD.
2. Draft initial data model (ERD).
3. Analyze data entities and relationships into final ERD.
4. Refine DFD according to the ERD data model (create logical process model).
5. Decompose logical process model into procedural elements.
6. Specify the details of each individual procedural element.

Object-oriented analysis and design



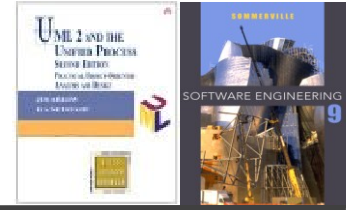
- Software engineering approach that models a system as a group of interacting objects.
- Each object represents some entity of interest in the system being modeled, and is characterized by its class, its state (data elements), and its behavior.
- Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects.
- There are a number of different methods, defining the ordering of modeling activities. The modeling notation uses to be unified (UML).

Object-oriented methods



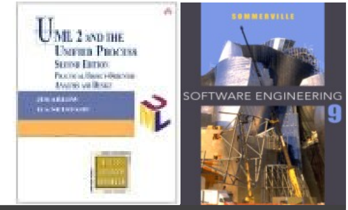
- ✧ Jim Rumbaugh: Object Modelling Technique (OMT)
- ✧ Coad-Yourdon: Method for Object-Oriented Analysis (OOA)
- ✧ Jacobson: Object-Oriented Software Engineering (OOSE)
- ✧ **Kruchten et al.: Rational Unified Process (RUP)**
  - Risk-driven iterations, component-based, with continuous quality verification and change management.
- ✧ **Booch-Jacobson-Rumbaugh: Unified Process (UP)**
  - Simplified non-commercial version of RUP maintained by Object Management Group (OMG).

UML notation for object-oriented methods



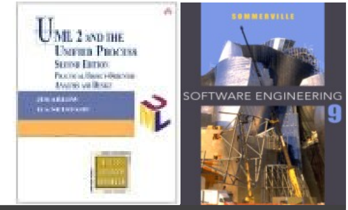
- External perspective
  - **Use case diagram**
- Structural perspective
  - **Class diagram**, Object diagram, Component diagram, Package diagram, Deployment diagram, Composite structure diagram
- Interaction perspective
  - **Sequence diagram**, Communication diagram, Interaction overview diagram, Timing diagram
- Behavioral perspective
  - **Activity diagram**, State diagram

Examplary method (Unified Process, analysis and design excerpt)



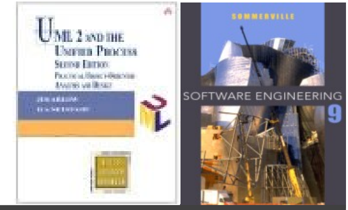
1. Requirements
   - System boundary, actors and requirements modelling with **Use Case diagram**.
2. Analysis
   - Identification of analysis classes, relationships, inheritance and polymorphism, and their documentation with a **Class diagram**.
   - Use Case realization with **Interaction** and **Activity diagrams**.
3. Design
   - Design classes, interfaces and components, resulting in refined **Class diagrams** and **Component diagrams**.
   - Detailed Use Case realization with **Interaction** and **State diagrams**.

Key points



- Structured methods
  - System as a set of nested processes accessing system data.
- Object-oriented methods
  - System as a set of interacting objects (functions and data).

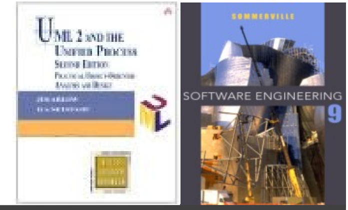
	Structured analysis	Object-oriented analysis
System boundary	Context diagram	Use case diagram
Functionality	Data flow diagram	Activity diagram Interaction diagrams
Data	Entity-relationship diagram	Class diagram Object diagram
Control	State diagram	State diagram



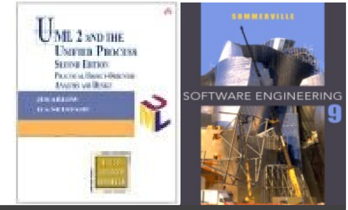
Object-Oriented Analysis

Lecture 4/Part 3

Analysis - purpose



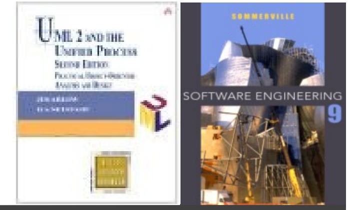
- Produce an Analysis Model of the system's desired behaviour:
  - This model should be a statement of what the system does not how it does it
  - We can think of the analysis model as a "first-cut" or "high level" design model
  - It is in the language of the business
- In the Analysis Model we identify:
  - Analysis classes
  - Use-case realizations



Object-Oriented Analysis: Objects and Classes

Lecture 4/Part 4

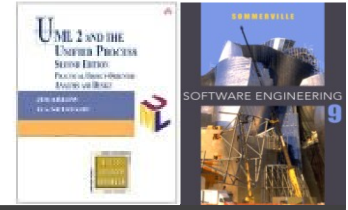
What are objects?



- Objects consist of data and function packaged together in a reusable unit. Objects *encapsulate* data
- Every object is an instance of some *class* which defines the common set of *features* (attributes and operations) shared by all of its instances. Objects have:
  - Attribute values – the data part
  - Operations – the behaviour part

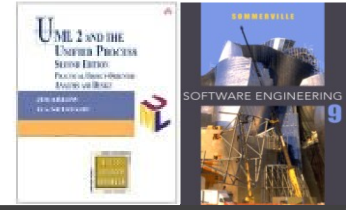
© Clear View Training 2010 v2.6

All objects have

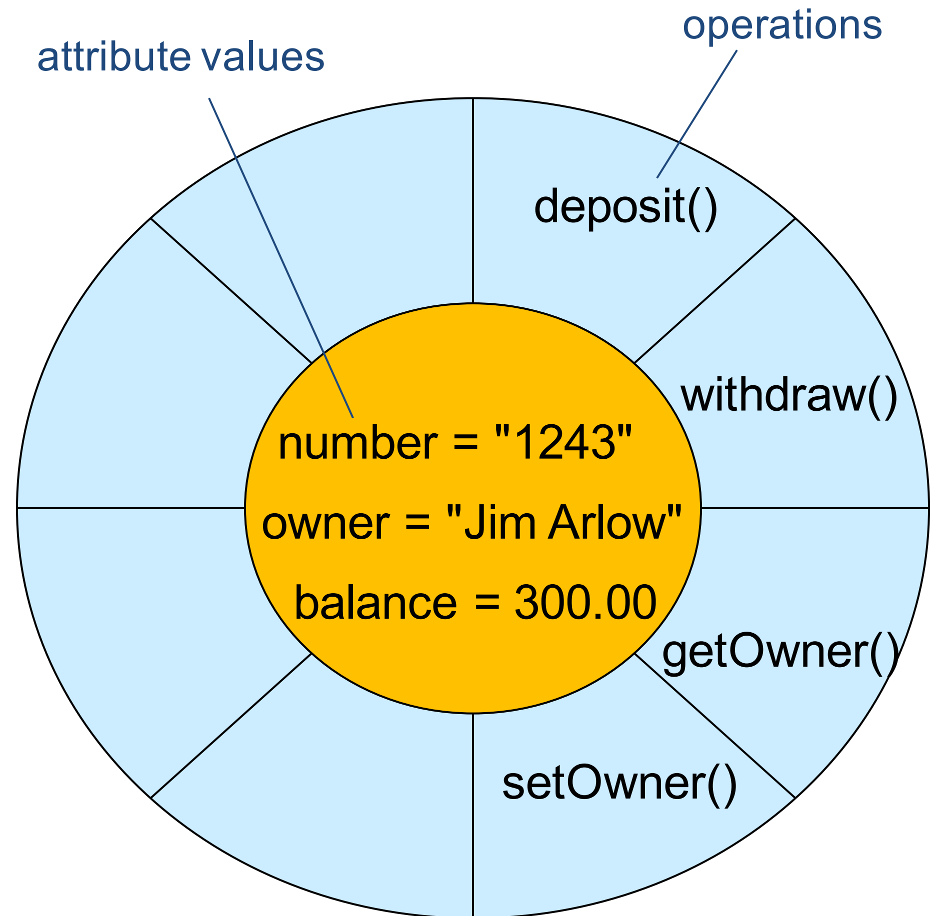


- *Identity*: Each object has its own unique identity and can be accessed by a unique handle
- *State*: This is the actual data values stored in an object at any point in time
- *Behaviour*: The set of operations that an object can perform

Encapsulation



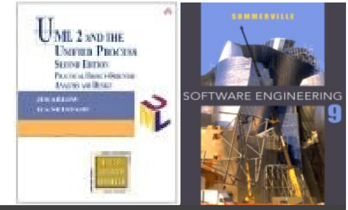
- Data is hidden inside the object. The only way to access the data is via one of the operations
- This is *encapsulation* or *data hiding* and it is a very powerful idea. It leads to more robust software and reusable code.



An Account Object

© Clear View Training 2010 v2.6

Messaging



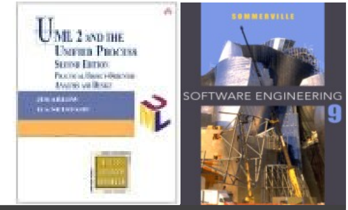
- In OO systems, objects send messages to each other over links
- These messages cause an object to invoke an operation

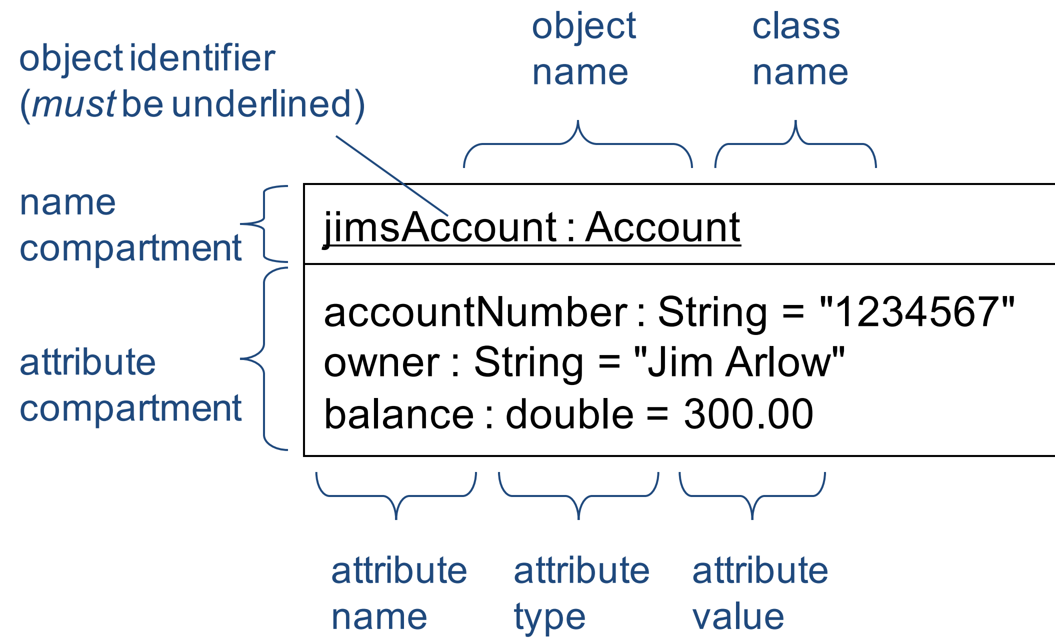
the Bank object sends the message "withdraw 150.00" to an Account object.

the Account object responds by invoking its withdraw operation. This operation decrements the account balance by 150.00.

© Clear View Training 2010 v2.6

UML Object Syntax





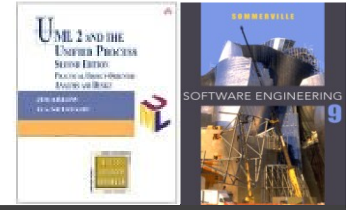
variants
(N.B. we've omitted the attribute compartment)

object and class name	jimsAccount : Account
object name only	jimsAccount
class name only	: Account

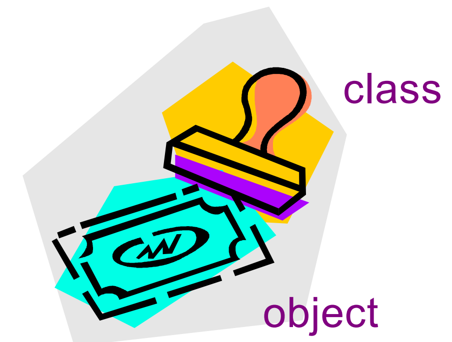
an anonymous object

- All objects of a particular class have the same set of operations. They are not shown on the object diagram, they are shown on the class diagram (see later)
- Attribute types are often omitted to simplify the diagram
- Naming: object and attribute names in lowerCamelCase, class names in UpperCamelCase

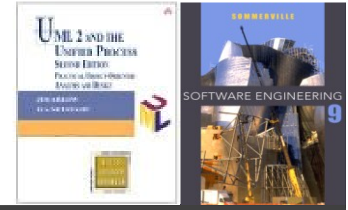
What are classes?

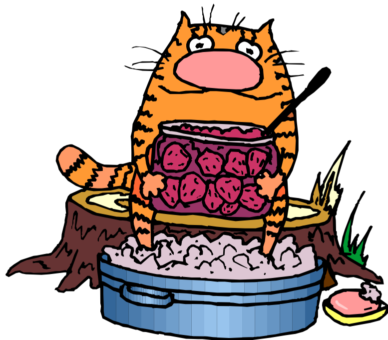


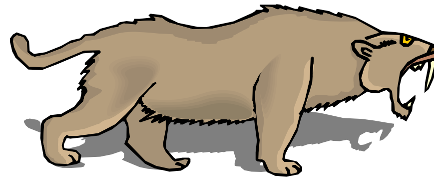
- Every object is an instance of one class - the class describes the "type" of the object
- Classes allow us to model sets of objects that have the *same* set of features - a class acts as a template for objects:
  - The class determines the structure (set of features) of all objects of that class
  - All objects of a class *must* have the same set of operations, *must* have the same attributes, but *may* have different attribute values
- Classification is one of the most important ways we have of organising our view of the world
- Think of classes as being like:
  - Rubber stamps
  - Cookie cutters

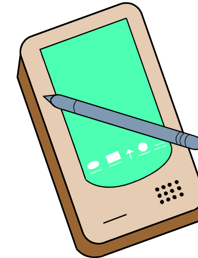


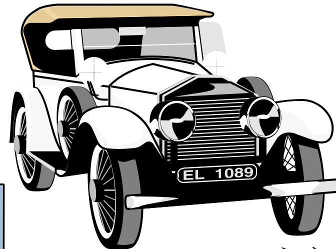
Exercise - how many classes?

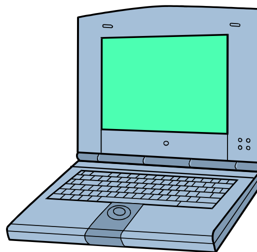


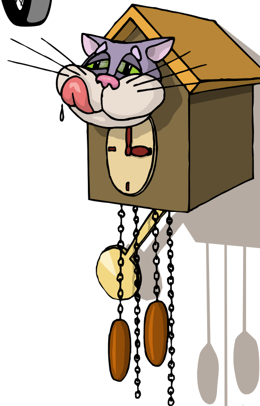






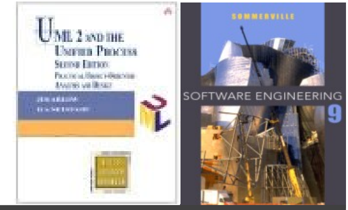






© Clear View Training 2010 v2.6

Classes and objects



- Objects are instances of classes
- Instantiation is the creation of new instances of model elements
- Most classes provide special operations called *constructors* to create instances of that class. These operations have class-scope i.e. they belong to the class itself rather than to objects of the class
- We will see instantiation used with other modelling elements later on

class

Account
accountNumber : String owner : String balance : double
withdraw() deposit()

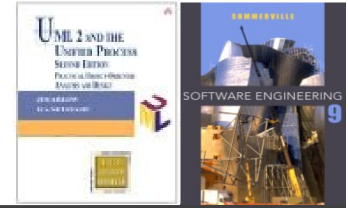
«instantiate» «instantiate» «instantiate»

<u>JimsAccount:Account</u>	<u>fabsAccount:Account</u>	<u>ilasAccount:Account</u>
accountNumber : "801" owner : "Jim" balance : 300.00	accountNumber : "802" owner : "Fab" balance : 1000.00	accountNumber : "803" owner : "Ila" balance : 310.00

objects

objects are instances of classes

UML class notation



class name

tagged values

name compartment

Window

{author = Jim,
status = tested}

attribute compartment

+size : Area=(100,100)
#visibility : Boolean = false
+defaultSize: Rectangle
#maximumSize : Rectangle
-xptr : XWindow*

initial values

visibility adornment

operation compartment

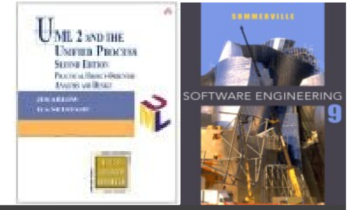
+create()
+hide()
+display(location : Point)
-attachXWindow(xwin : XWindow*)

class scope (static) operation

- Classes are named in UpperCamelCase
- Use descriptive names that are nouns or noun phrases
- Avoid abbreviations!

© Clear View Training 2010 v2.6

Attribute compartment

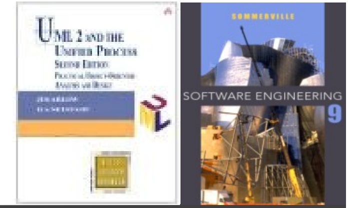


visibility name : type multiplicity = initialValue

mandatory

- Everything is optional except name
- initialValue is the value the attribute gets when objects of the class are instantiated
- Attributes are named in lowerCamelCase
  - Use descriptive names that are nouns or noun phrases
  - Avoid abbreviations
- Attributes may be prefixed with a stereotype and postfixed with a list of tagged values

Visibility

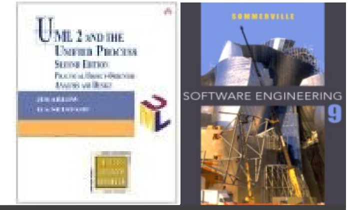


Symbol	Name	Semantics
+	public	Any element that can access the class can access any of its features with public visibility
-	private	Only operations within the class can access features with private visibility
#	protected	Only operations within the class, or within children of the class, can access features with protected visibility
~	package	Any element that is in the same package as the class, or in a nested subpackage, can access any of its features with package visibility

PersonDetails
-name : String [2..*] -address : String [3] -emailAddress : String [0..1]

- You may ignore visibility in analysis
- In design, attributes usually have private visibility (encapsulation)

Multiplicity



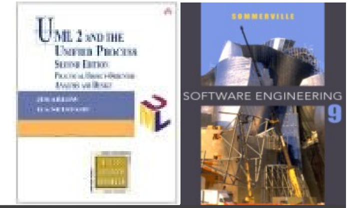
- ✧ Multiplicity allows you to model collections of things
  - [0..1] means an that the attribute may have the value null

PersonDetails
-name : String [2..*] -address : String [3] -emailAddress : String [0..1]

name is composed of 2 or more Strings
address is composed of 3 Strings
emailAddress is composed of 1 String or null

multiplicity expression

Operation compartment



operation signature

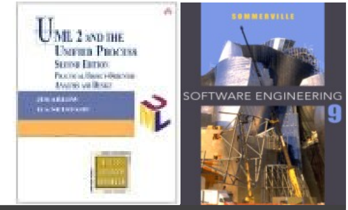
visibility name(direction parameterName: parameterType = default, …) : returnType

parameter list

there may be a comma delimited list of return types - r1, r2,… rn

- Operations are named lowerCamelCase
  - Special symbols and abbreviations are avoided
  - Operation names are usually a verb or verb phrase
- Operations may have more than one returnType
  - They can return multiple objects (see next slide)
- Operations may be prefixed with a stereotype and postfixed with a list of tagged values

Parameter direction



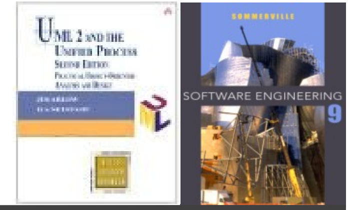
parameter direction	semantics
in	the parameter is an input to the operation. It is not changed by the operation. This is the default
out	the parameter serves as a repository for output from the operation
inout	the parameter is an input to the operation and it may be changed by the operation
return	the parameter is one of the return values of the operation. An alternative way of specifying return values

use in detailed design only!

example of multiple return values:

```
maxMin( in a: int, in b:int, return maxValue:int return minValue:int )
max, min = maxMin( 5, 10 )
```

Scope



- There are two kinds of scope for attributes and operations:

BankAccount

-accountNumber : int
-count : int = 0

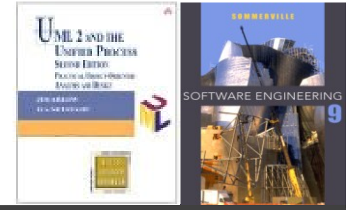
+create(aNumber : int)
+getNumber() : int
-incrementCount()
+getCount() : int

class scope (underlined)

instance scope (the default)

© Clear View Training 2010 v2.6

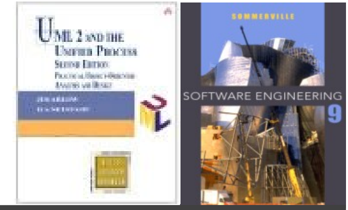
Instance scope vs. class scope



	instance scope	class scope
attributes	By default, attributes have instance scope	Attributes may be defined as class scope
	Every object of the class gets its own copy of the instance scope attributes	Every object of the class shares the same, single copy of the class scope attributes
	Each object may therefore have different instance scope attribute values	Each object will therefore have the same class scope attribute values
operations	By default, operations have instance scope	Operations may be defined as class scope
	Every invocation of an instance scope operation applies to a specific instance of the class	Invocation of a class scope operation does not apply to any specific instance of the class – instead, you can think of class scope operations as applying to the class itself
	You can't invoke an instance scope operation unless you have an instance of the class available. You can't use an instance scope operation of a class to create objects of that class, as you could never create the first object	You can invoke a class scope operation even if there is no instance of the class available – this is ideal for object creation operations

scope determines access

Object construction



- How do we create instances of classes?
- Each class defines one or more class scope operations which are *constructors*. These operations create new instances of the class

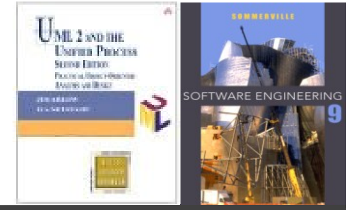
BankAccount
+create(aNumber : int)

generic constructor name

BankAccount
+BankAccount(aNumber : int)

Java/C++ standard

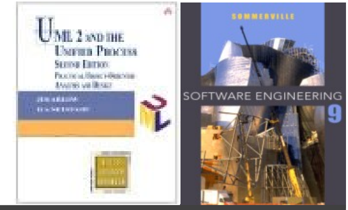
ClubMember class example



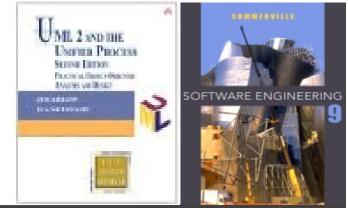
- Each ClubMember object has its own copy of the attribute membershipNumber
- The numberOfMembers attribute exists only once and is shared by all instances of the ClubMember class
- Suppose that in the create operation we increment numberOfMembers:
  - What is the value of count when we have created 3 account objects?

ClubMember
-membershipNumber : String -memberName : String -numberOfMembers : int = 0
+create(number : String, name : String) +getMembershipNumber() : String +getMemberName() : String -incrementNumberOfMembers() +decrementNumberOfMembers() +getNumberOfMembers() : int

Key points



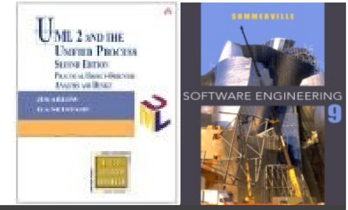
- We have looked at objects and classes and examined the relationship between them
- We have explored the UML syntax for modelling classes including:
  - Attributes
  - Operations
- We have seen that scope controls access
  - Attributes and operations are normally instance scope
  - We can use class scope operations for constructor and destructors
  - Class scope attributes are shared by all objects of the class and are useful as counters



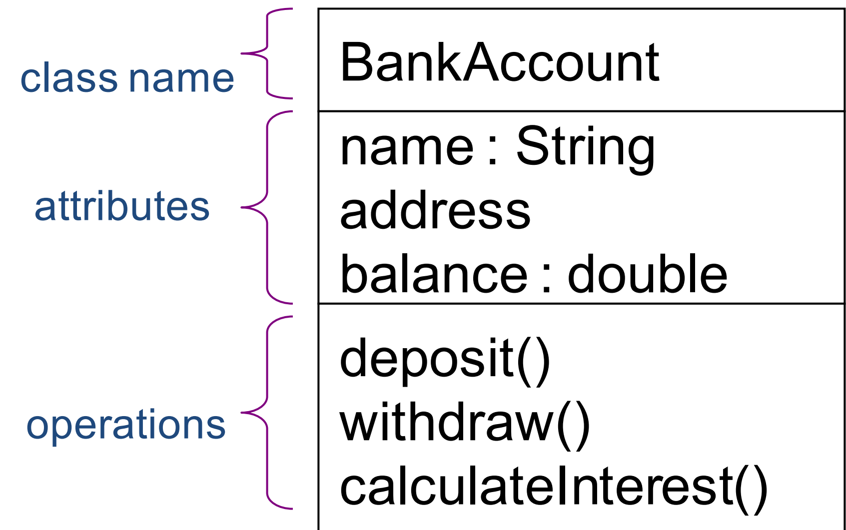
Object-Oriented Analysis: Finding Analysis Classes

Lecture 4/Part 5

What are Analysis classes?

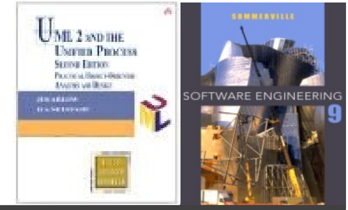


- ✧ Analysis classes represent a crisp abstraction in the problem domain
  - They may ultimately be refined into one or more design classes
- ✧ All classes in the Analysis model should be Analysis classes
- ✧ Analysis classes have:
  - A very "high level" set of attributes. They *indicate* the attributes that the design classes *might* have.
  - Operations that specify at a high level the key services that the class must offer. In Design, they will become actual, implementable, operations.
- ✧ Analysis classes must map onto real-world business concepts



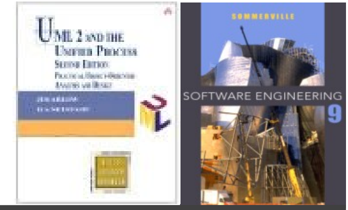
We always specify attribute types if we know what they are!

What makes a good analysis class?



- ✧ Its name reflects its intent
- ✧ It is a *crisp abstraction* that models one specific element of the problem domain
  - It maps onto a clearly identifiable feature of the problem domain
- ✧ It has *high cohesion*
  - Cohesion is the degree to which a class models a single abstraction
  - Cohesion is the degree to which the *responsibilities* of the class are semantically related
- ✧ It has *low coupling*
  - Coupling is the degree to which one class depends on others

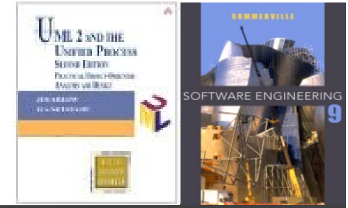
Rules of thumb



- 3 to 5 responsibilities per class
- Each class collaborates with others
- Beware many very small classes
- Beware few but very large classes
- Beware of “functoids”
- Beware of “omnipotent” classes
- Avoid deep inheritance trees

A *responsibility* is a contract or obligation of a class - it resolves into operations and attributes

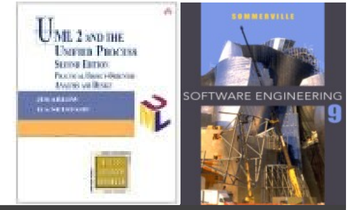
Finding classes



- Perform noun/verb analysis on documents:
  - Nouns are candidate classes
  - Verbs are candidate *responsibilities*
- Perform CRC card analysis
  - A brainstorming technique using sticky notes
  - Useful for brainstorming, Joint Application Development (JAD) and Rapid Application development (RAD)
- With both techniques, beware of spurious classes:
  - Look for *synonyms* - different words that mean the same
  - Look for *homonyms* - the same word meaning different things
- Look for "hidden" classes!
  - Classes that don't appear as nouns or as cards

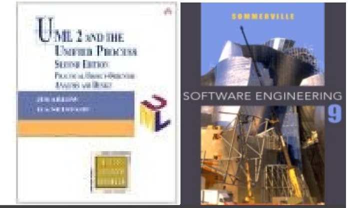
© Clear View Training 2010 v2.6

Noun/verb analysis procedure



- Collect all of the relevant documentation
  - Requirements document
  - Use cases
  - Project Glossary
  - Anything else!
- Make a list of nouns and noun phrases
  - These are candidate classes or attributes
- Make a list of verbs and verb phrases
  - These are candidate responsibilities
- Tentatively assign attributes and responsibilities to classes

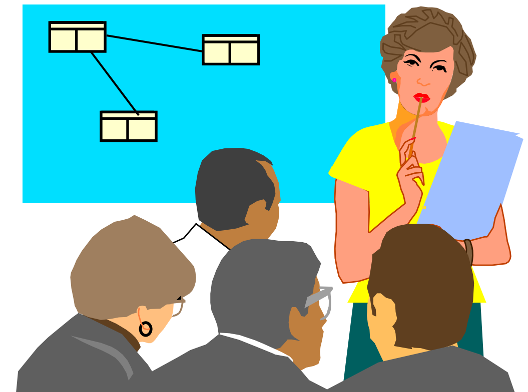
CRC card procedure



Class Name: BankAccount	
Responsibilities:	Collaborators:
Maintain balance	Bank

things the class does

things the class works with

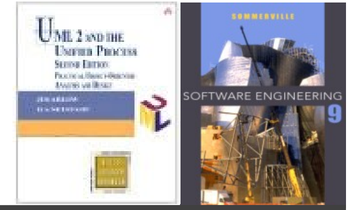


✧ Class, Responsibilities and Collaborators

✧ Separate information collection from information analysis

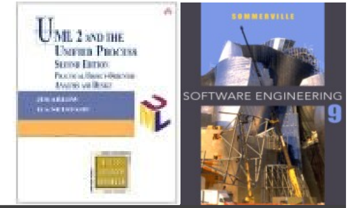
- Part 1: Brainstorm
  - *All* ideas are good ideas in CRC analysis
  - Never argue about something – write it down and analyse it later!
- Part 2: Analyse information - consolidate with noun/verb

Other sources of classes



- ✧ Physical objects
- ✧ Paperwork, forms etc.
  - Be careful with this one – if the existing business process is very poor, then the paperwork that supports it might be irrelevant
- ✧ Known interfaces to the outside world
- ✧ Conceptual entities that form a cohesive abstraction e.g. LoyaltyProgramme

Key points



- We've looked at what constitutes a well-formed analysis class
- We have looked at two analysis techniques for finding analysis classes:
  - Noun verb analysis of use cases, requirements, glossary and other relevant documentation
  - CRC analysis