

Object-oriented programming

Rafał Witkowski

2021



Mikołaj K [REDACTED]

Poza tym nie wiem czy słyszeliście państwo o takiej technice jak programowanie obiektowe - chodzi z grubsza o to, ze obiekty którymi otacza się ludzi mają wpływ na nasze mózgi tak żebyśmy byli zniewoleni.

Programowanie obiektowe nawet wykorzystuje swoją specjalną technikę która nazywa się Jawa - przez to ludzie myślą że simulacja na która patrzą to tak naprawdę java czyli jest prawdziwa

JOE MONSTER

Przed chwilą Lubię to! Więcej

Introduction

- Goal for the lecture:
 - Introduction to object-oriented programming,
 - Introduction to the use of patterns.
 - Introduction to UML (Unified Modelling Language),
 - Skills acquired during the classes will allow to create simple object-oriented software
- Prerequisites for a lecture listener:
 - Knowledge of some object-oriented programming languages (C++, Java, C#) and experience in programming in one language, with procedures and function

Why object-oriented programming is not easy?

- Knowledge of syntax is not sufficient to effectively program using object-oriented languages (C++, Java, C#) - "Having a hammer does not make us immediately architect".
- Programming in these languages becomes effective in case of correct "object-oriented thinking". Therefore it is necessary before starting the programming, carrying out an analysis (object-oriented analysis and design: OOA/D).
- Critical and basic skills of an OOA/D is the ability to correspond to individual responsibility, deciding how the responsibility should interact with each other, defining what should do what.

Why object-oriented programming?

- Real World Modelling:

Object-oriented programming means building in the program models of objects from the real world. The programmer's work consists of the revitalisation of these objects and making them to start cooperate with each other.

Why object-oriented programming?

- Code reusability:

Once written and saved, the code can be used several times. The following is implemented it is thanks to inheritance (methods from the superclass are available in classes derivatives).

Why object-oriented programming?

- Extensibility:

Thanks to polymorphism, implementation of new classes does not require to modify parts of the code already done. Polymorphic methods are responsible for carrying out activities related to a given class. If new classes are introduced, appropriate polymorphic methods will be introduced.

Why object-oriented programming?

- Improved error detection:

Thanks to the connection of methods to objects, it is not possible to make errors of incorrect function calling with unauthorized arguments (methods operate on data related to the object). Thus, the number of errors of incorrect use of the function is minimized.

Why object-oriented programming?

- Simplicity in programming the user interfaces:

Object-oriented programming directly responds to the needs of creating "window" user interests. Windows can be seen as objects, consisting of other objects (buttons, menu items, etc.).

Object-oriented analysis

- The goal of object-oriented analysis is to provide answers to the question: how should the system work?
- Tasks carried out during the analysis:
 - creation of a logic system model describing the way of implementation by the system of set requirements (the logical model omits most of the implementation details),
 - to define a basic "dictionary" of domain knowledge in order to facilitate the analysis and subsequent development of applications.
- This is achieved by finding and describing the objects, understood as concepts
- Created models are saved with the use of notations defined in modelling languages (e.g. UML).

Object-oriented designing

- The aim of object-oriented designing is to answer the question: how to implement the system?
- Tasks carried out during the design process:
 - development of a detailed description,
 - definition of objects used in the program.
- This is achieved by defining the principles of objects cooperation in such a way that it meets the requirements (defining attributes, methods, etc.).
- The structure of the created software should as far as possible preserve the general structure of the model created in the previous phase.
- In the design phase the same notation is used as in the analysis phase.

Object-oriented programming

- Implementation (encoding) of the software project in the selected environment:
 - use of the object-oriented language,
 - reuse of already existing object libraries,
 - use of tools for quick application development,
 - use of code generators.
- Take steps to develop reliable software:
 - avoidance of dangerous techniques,
 - the need-to-know principle (you know only what you need to know),
 - the use of type-checking compilers.

UML – object-oriented models

- What is a model?
 - A model is a simplification of reality.
- Why do we model?
 - We build models to better understand the system we create.

UML – object-oriented models

- In what modelling helps us in:
 - in the visualization of the system as it is, or as it should be,
 - in the specification of the structure or behaviour of the system,
 - serves as a template during application development,
 - documents the results of the work carried out.
- Why do we need formal models?
 - A unified language will facilitate communication and allow to describe the created model in a uniform way.

UML – what is it?

- Unified Modelling Language
- A unified language for the object-oriented modelling.
- UML provides the system designer a tool for visualizing, specifying, constructing and documenting concepts and mechanisms that make up the solution of the problem.

UML characteristic

- UML is developed by the following persons: Grady Booch, James Rumbaugh and Ivar Jacobson,
- successor of other object-oriented methodologies such as OMT (Rumbaugh), OOA/OOD (Coad, Yourdon), OOAD (Booch), Objectory (Jacobson),
- a combination of theory and practice: Rational Software Corporation.

Patterns

- „Design templates“ are a written collection of advanced experience of object-oriented program designers.
- Design patterns are written in a codified way that allows to describe a programming problem and its solution.
- The most famous design patterns (23 patterns) were developed by the so-called Gang of Four (Gamma, Helm, Johnson and Vlissides):
 - Adapter,
 - Factory,
 - Singleton,
 - Strategy,
 - – ...

Bibliography

1. Craig Larman, Applying UML and Patterns An Introduction to Object-Oriented Analysis and Design and the Unified Process, Prentice Hall, 2002
2. Rebecca Wirfs-Brock, Alan McKean, Object Design – Roles, Responsibilities and Collaborations, Addison Wesley, 2003
3. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 1995
4. Richard C. Lee, William M. Tepfenhart, UML and C++ A practical guide to object-oriented development, Prentice Hall, 1997
5. Bruce Eckel, Thinking in Java

Introduction to UML

Team Emertxe



What is UML?

- Unified Modeling Language
 - OMG Standard, Object Management Group
 - Based on work from Booch, Rumbaugh, Jacobson
- UML is a modeling language to express and design documents, software
 - Particularly useful for OO design
 - Not a process, but some have been proposed using UML
 - Independent of implementation language

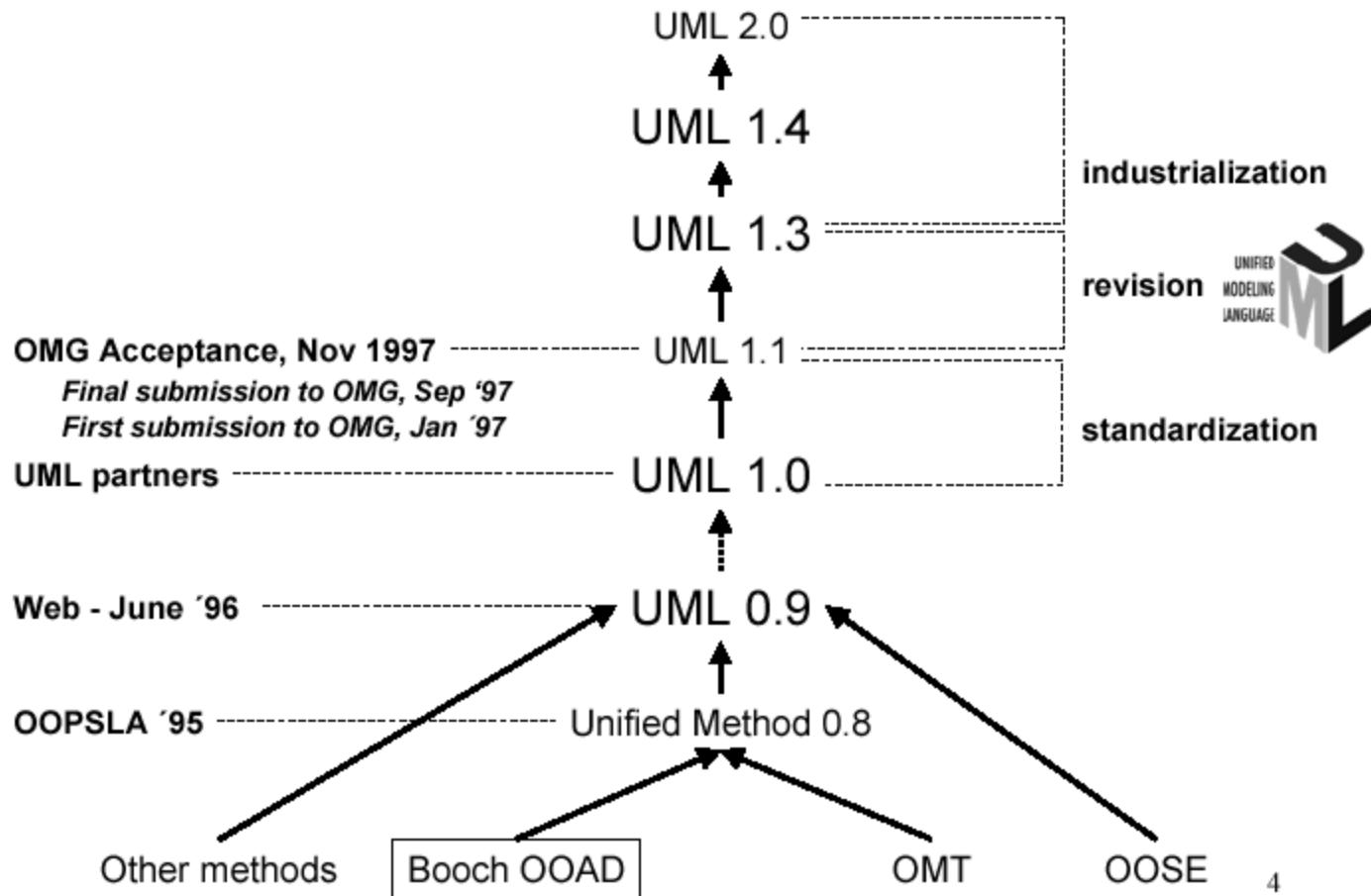
Why use UML

- Open Standard, Graphical notation for
 - Specifying, visualizing, constructing, and documenting software systems
- Language can be used from general initial design to very specific detailed design across the entire software development lifecycle
- Increase understanding/communication of product to customers and developers
- Support for diverse application areas
- Support for UML in many software packages today (e.g. Rational, plugins for popular IDE's like NetBeans, Eclipse)
- Based upon experience and needs of the user community

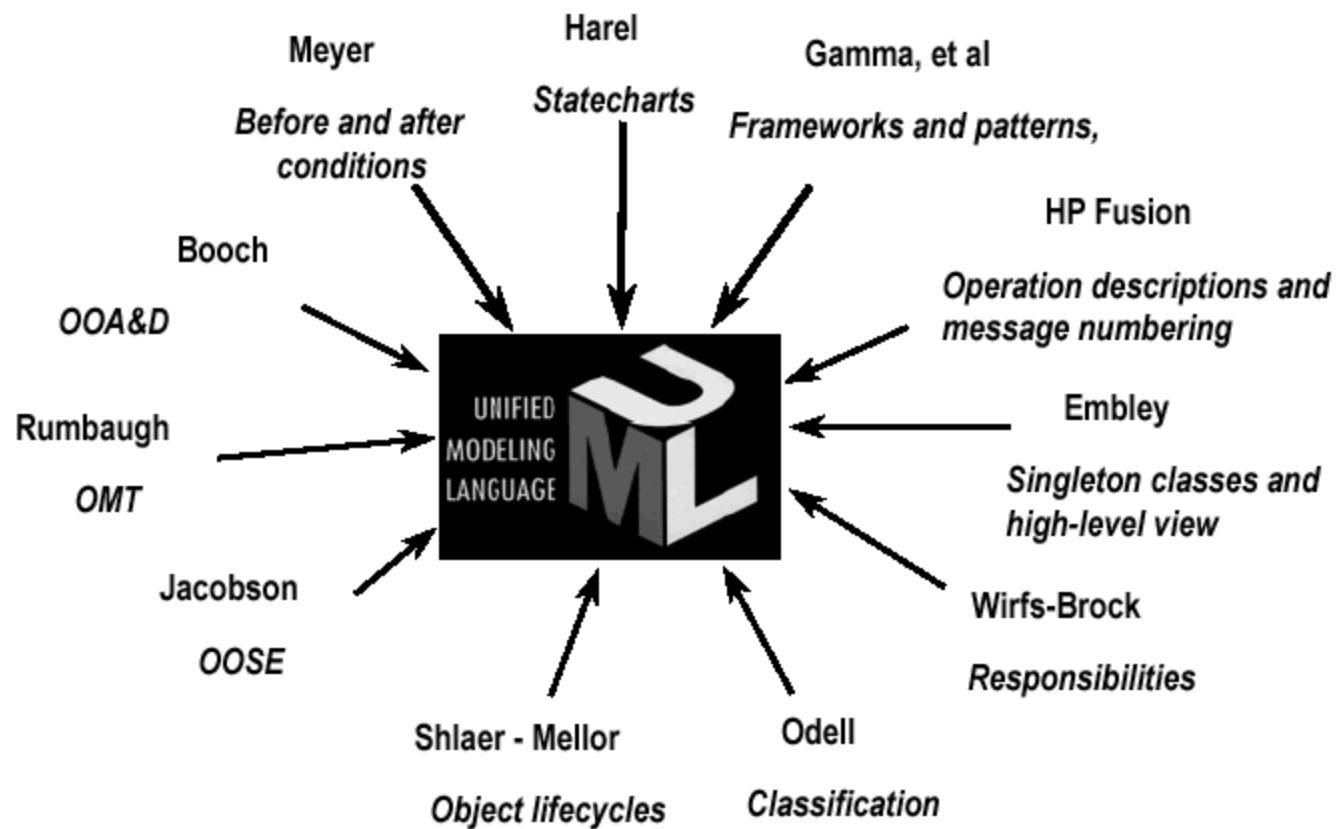
Brief History

- Inundated with methodologies in early 90's
 - Booch, Jacobson, Yourden, Rumbaugh
- Booch, Jacobson merged methods 1994
- Rumbaugh joined 1995
- 1997 UML 1.1 from OMG includes input from others, e.g. Yourden
- UML v2.0 current version

History of UML



Contributions to UML



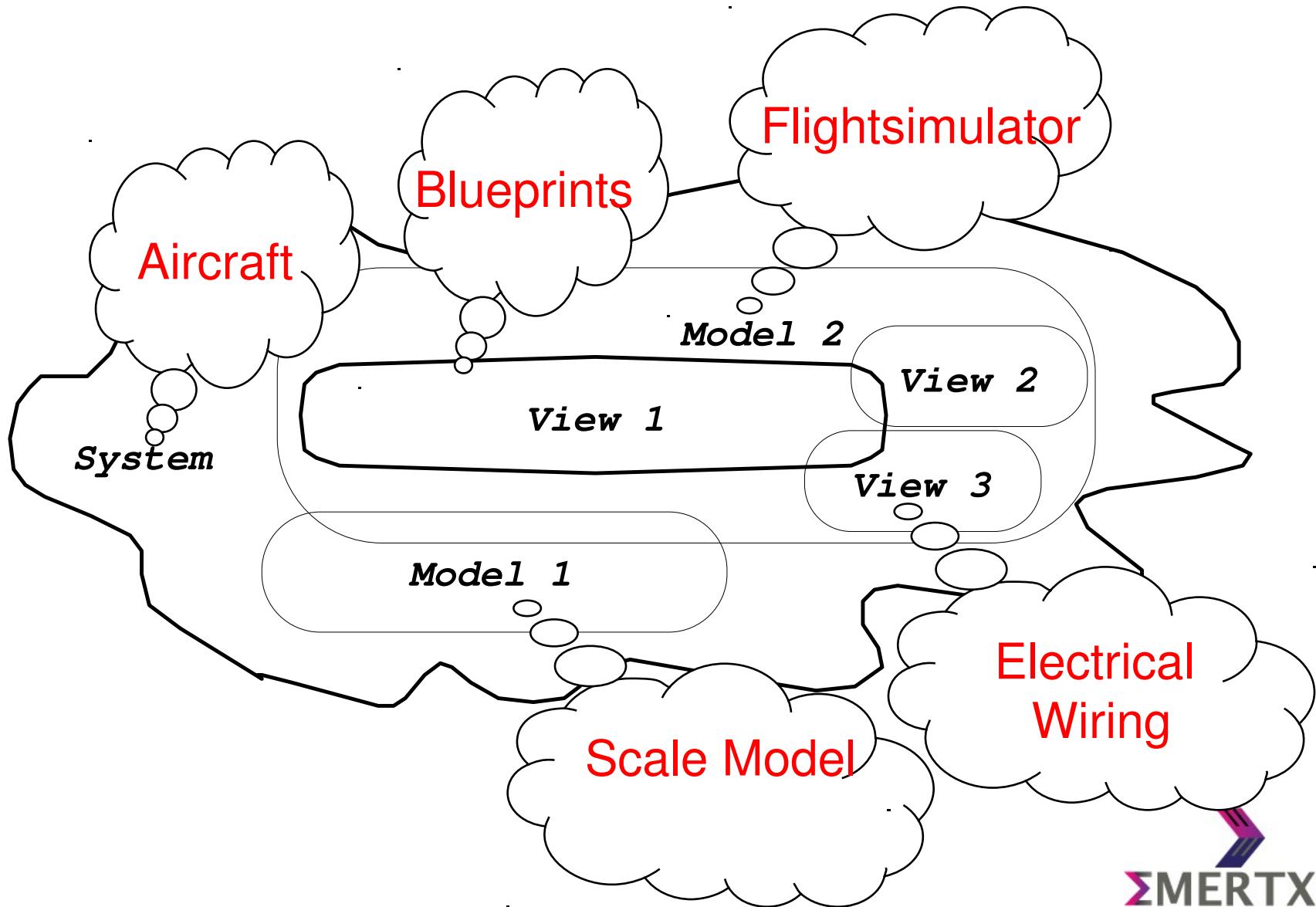
Systems, Models and Views

- A ***model*** is an abstraction describing a subset of a system
- A ***view*** depicts selected aspects of a model
- A ***notation*** is a set of graphical or textual rules for depicting views
- Views and models of a single system may overlap each other

Examples:

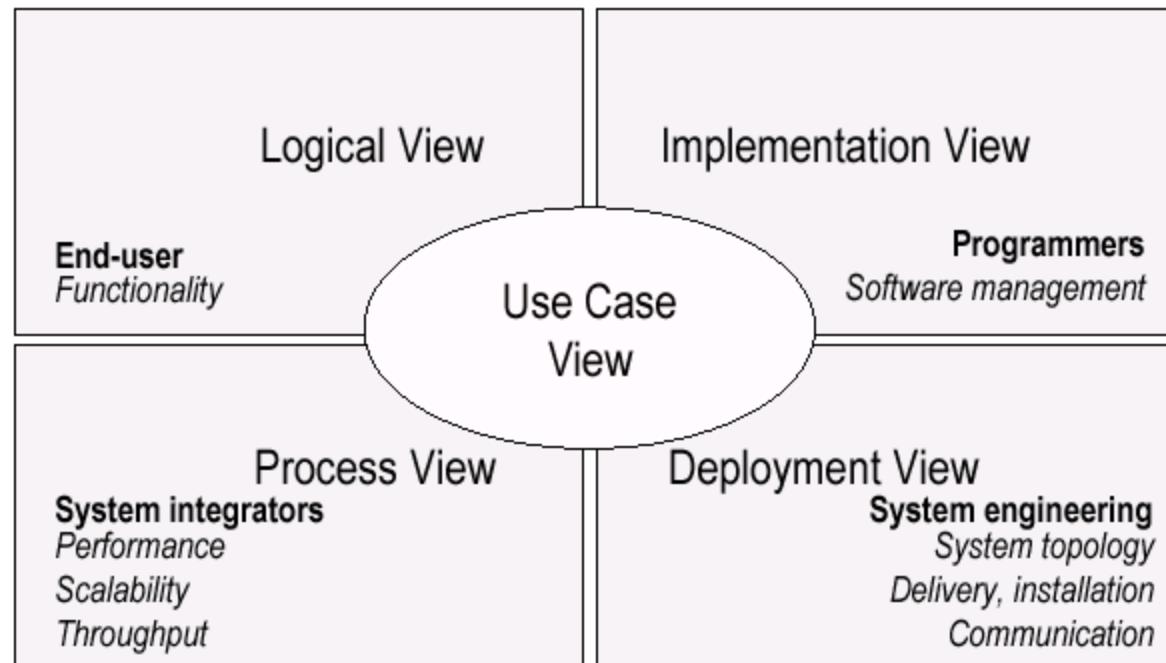
- System: Aircraft
- Models: Flight simulator, scale model
- Views: All blueprints, electrical wiring, fuel system

Systems, Models and Views

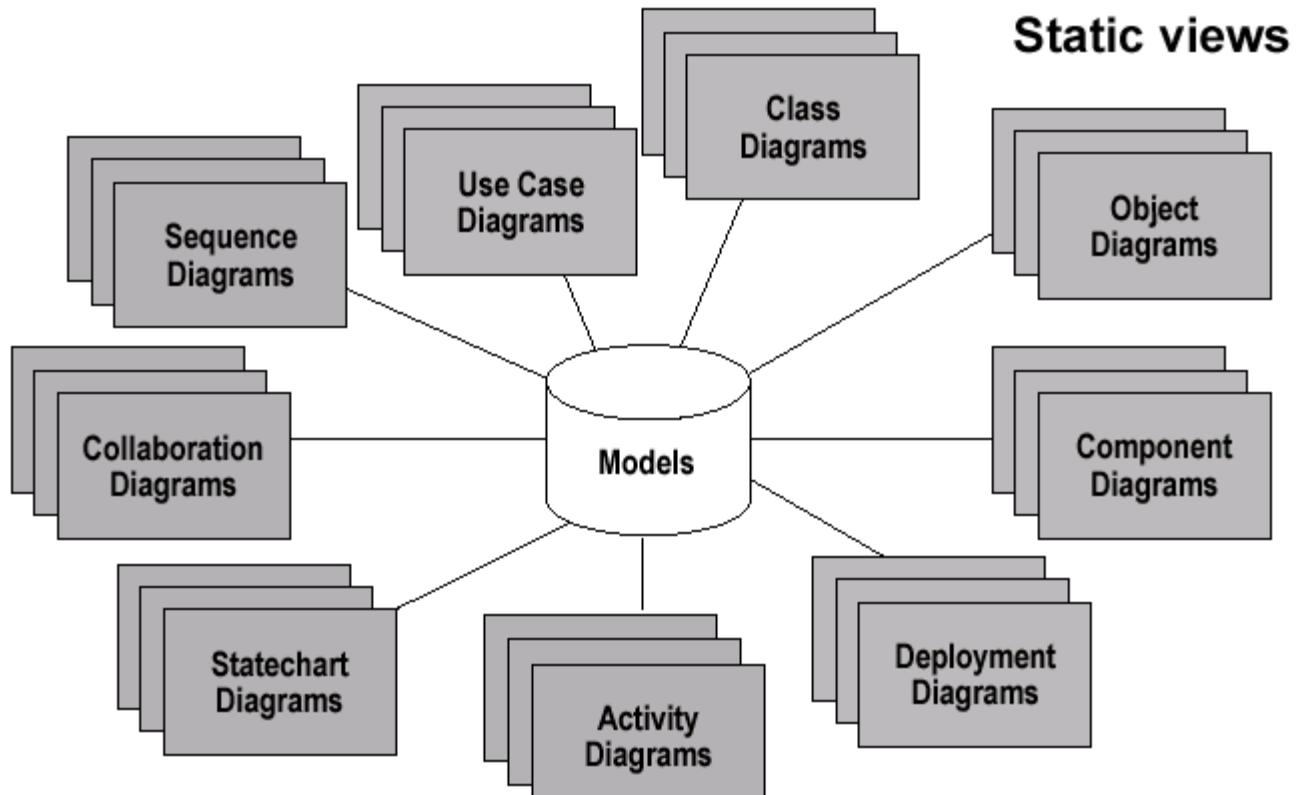


Models, Views, Diagrams

- UML is a multi-diagrammatic language
 - Each diagram is a view into a model
 - Diagram presented from the aspect of a particular stakeholder
 - Provides a partial representation of the system
 - Is semantically consistent with other views
 - Example views



Models, Views, Diagrams



Dynamic views

How Many Views?

- Views should fit the context
 - Not all systems require all views
 - Single processor: drop deployment view
 - Single process: drop process view
 - Very small program: drop implementation view
- A system might need additional views
 - Data view, security view, ...

UML: First Pass

- You can model 80% of most problems by using about 20 % UML
- We only cover the 20% here

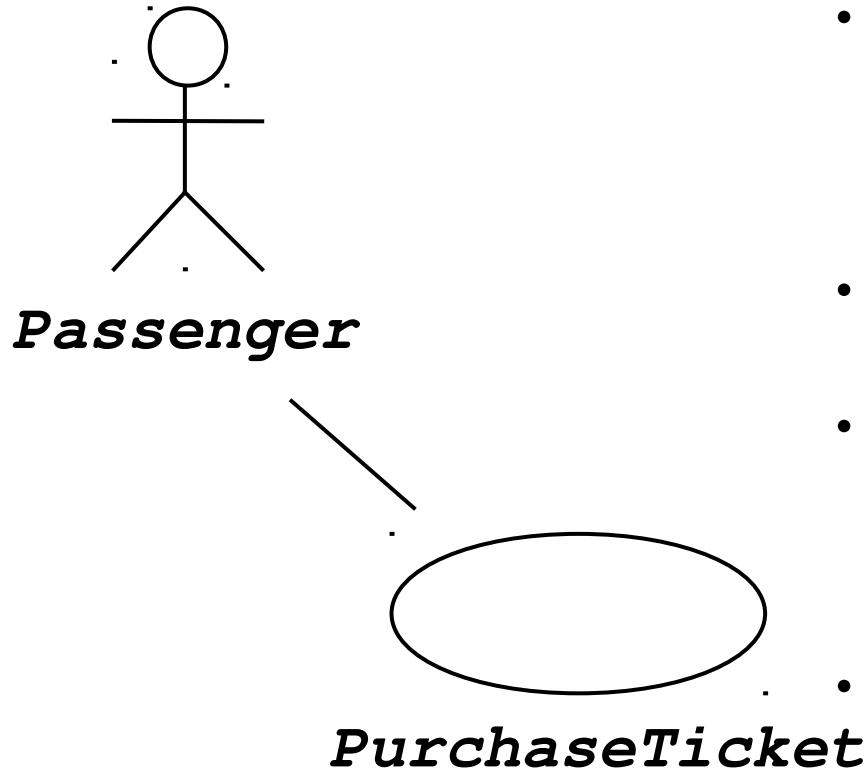
Basic Modeling Steps

- Use Cases
 - Capture requirements
- Domain Model
 - Capture process, key classes
- Design Model
 - Capture details and behaviors of use cases and domain objects
 - Add classes that do the work and define the architecture

UML Baseline

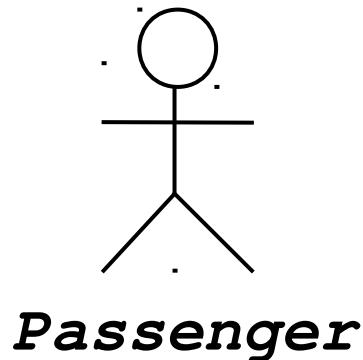
- Use Case Diagrams
- Class Diagrams
- Package Diagrams
- Interaction Diagrams
 - Sequence
 - Collaboration
- Activity Diagrams
- State Transition Diagrams
- Deployment Diagrams

Use Case Diagrams



- Used during requirements elicitation to represent external behavior
- *Actors* represent roles, that is, a type of user of the system
- *Use cases* represent a sequence of interaction for a type of functionality; summary of scenarios
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

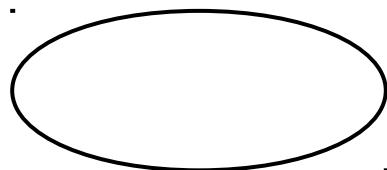
Actors



- An actor models an external entity which communicates with the system:
 - User
 - External system
 - Physical environment
- An actor has a unique name and an optional description.
- Examples:
 - Passenger: A person in the train
 - GPS satellite: Provides the system with GPS coordinates

Use Case

A use case represents a class of functionality provided by the system as an event flow.



PurchaseTicket

A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

Use Case Diagram: Example

Name: **Purchase ticket**

Participating actor: **Passenger**

Entry condition:

- **Passenger** standing in front of ticket distributor.
- **Passenger** has sufficient money to purchase ticket.

Exit condition:

- **Passenger** has ticket.

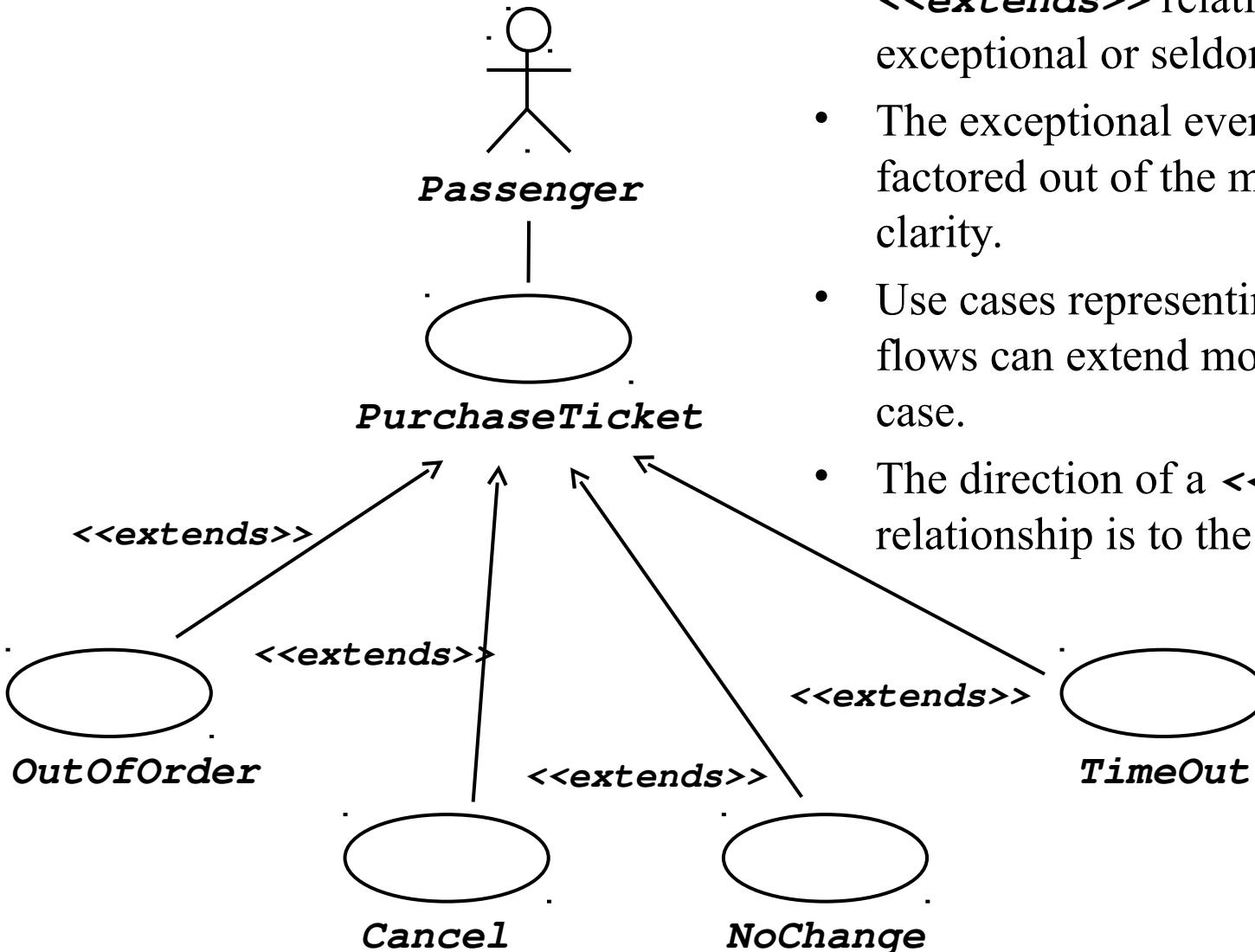
Event flow:

1. **Passenger** selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. **Passenger** inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

Anything missing?

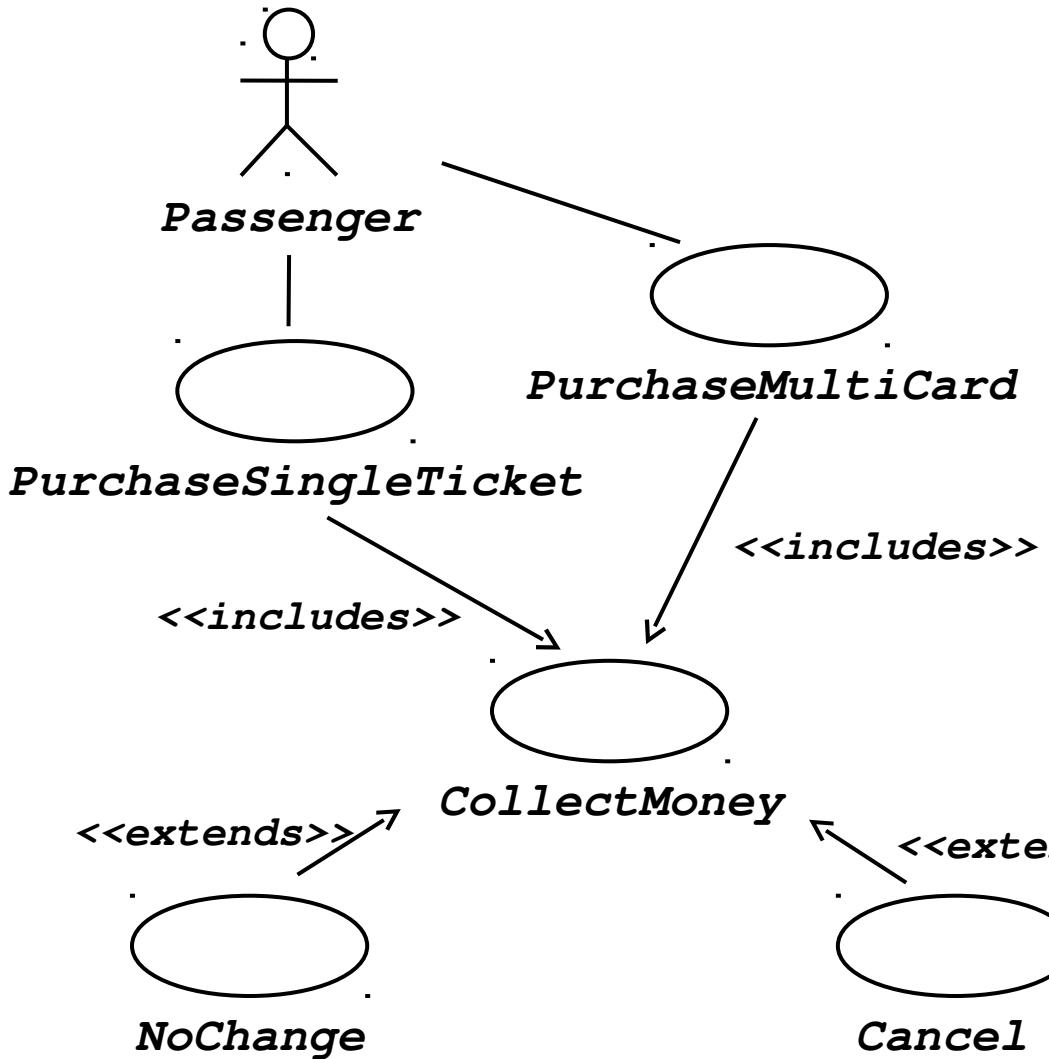
Exceptional cases!

The <<extends>> Relationship



- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case

The <<includes>> Relationship



- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

Use Cases are useful to...

- Determining requirements
 - New use cases often generate new requirements as the system is analyzed and the design takes shape.
- Communicating with clients
 - Their notational simplicity makes use case diagrams a good way for developers to communicate with clients.
- Generating test cases
 - The collection of scenarios for a use case may suggest a suite of test cases for those scenarios.

Use Case Diagrams: Summary

- Use case diagrams represent external behavior
- Use case diagrams are useful as an index into the use cases
- Use case descriptions provide meat of model, not the use case diagrams.
- All use cases need to be described for the model to be useful.

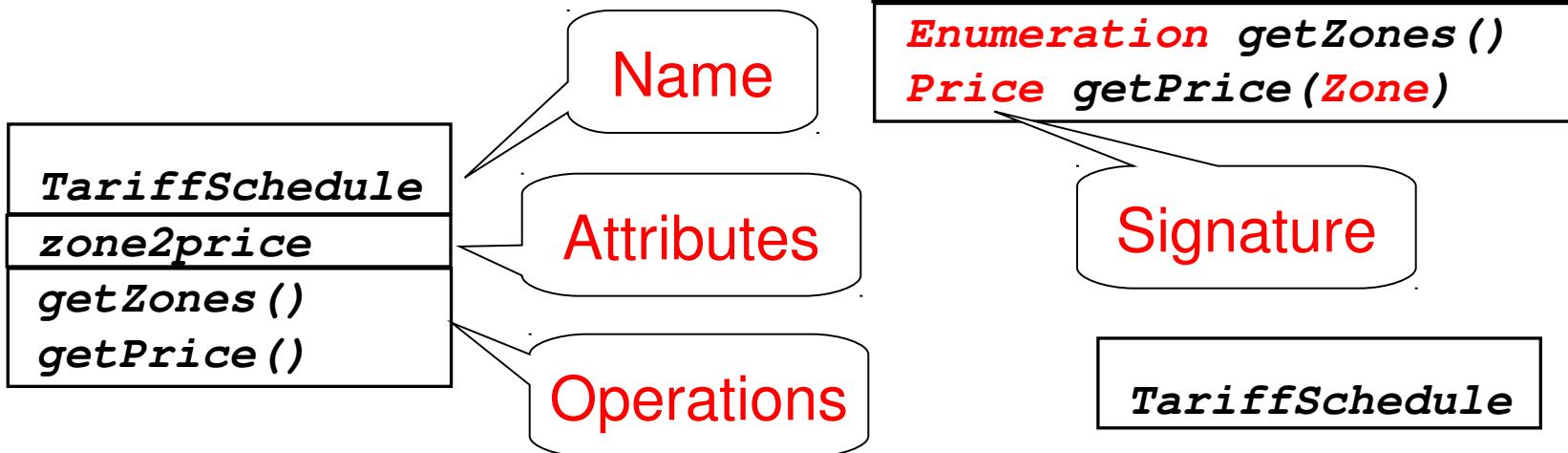
Class Diagrams

- Gives an overview of a system by showing its classes and the relationships among them.
 - Class diagrams are static
 - they display what interacts but not what happens when they do interact
- Also shows attributes and operations of each class
- Good way to describe the overall architecture of system components

Class Diagram: Perspectives

- We draw Class Diagrams under three perspectives
 - Conceptual
 - Software independent
 - Language independent
 - Specification
 - Focus on the interfaces of the software
 - Implementation
 - Focus on the implementation of the software

Classes: Not Just for Code



- A *class* represent a concept
- A class encapsulates state (*attributes*) and behavior (*operations*).
- Each attribute has a *type*.
- Each operation has a *signature*.
- The class name is the only mandatory information.

Instances

```
tarif 1974:TariffSchedule
```

```
zone2price = {  
  {'1', .20},  
  {'2', .40},  
  {'3', .60}}
```

- An *instance* represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their *values*.

UML Class Notation

- A class is a rectangle divided into three parts
 - Class name
 - Class attributes (i.e. data members, variables)
 - Class operations (i.e. methods)
- Modifiers
 - Private: -
 - Public: +
 - Protected: #
 - Static: Underlined (i.e. shared among all members of the class)
- Abstract class: Name in italics

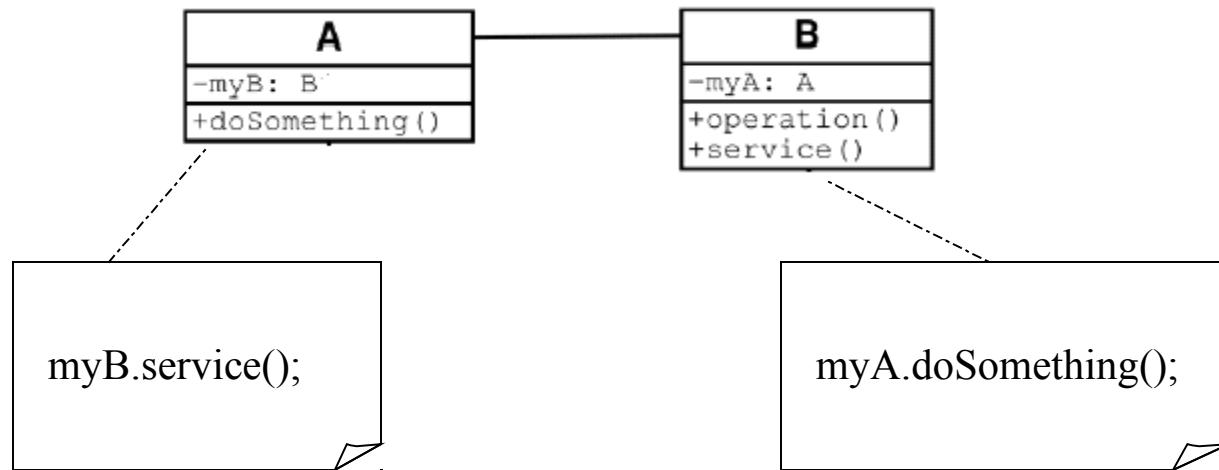
Employee
-Name : string +ID : long #Salary : double
+getName() : string +setName() -calcInternalStuff (in x : byte, in y : decimal)

UML Class Notation

- Lines or arrows between classes indicate relationships
 - Association
 - A relationship between instances of two classes, where one class must know about the other to do its work, e.g. client communicates to server
 - indicated by a straight line or arrow
 - Aggregation
 - An association where one class belongs to a collection, e.g. instructor part of Faculty
 - Indicated by an empty diamond on the side of the collection
 - Composition
 - Strong form of Aggregation
 - Lifetime control; components cannot exist without the aggregate
 - Indicated by a solid diamond on the side of the collection
 - Inheritance
 - An inheritance link indicating one class a superclass relationship, e.g. bird is part of mammal
 - Indicated by triangle pointing to superclass

Binary Association

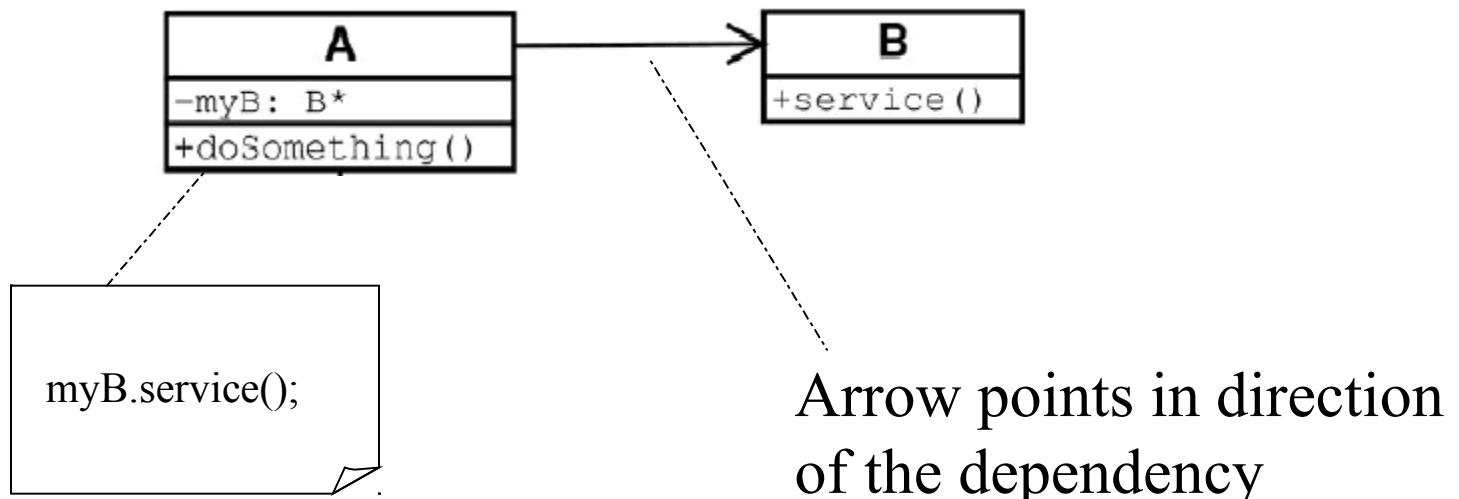
Binary Association: Both entities “Know About” each other



Optionally, may create an Associate Class

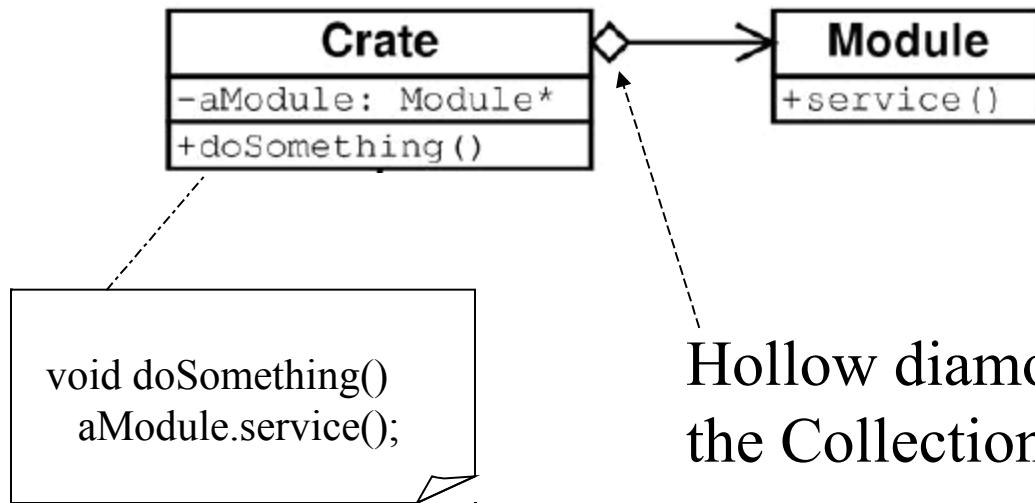
Unary Association

A knows about B, but B knows nothing about A



Aggregation

Aggregation is an association with a “collection-member” relationship



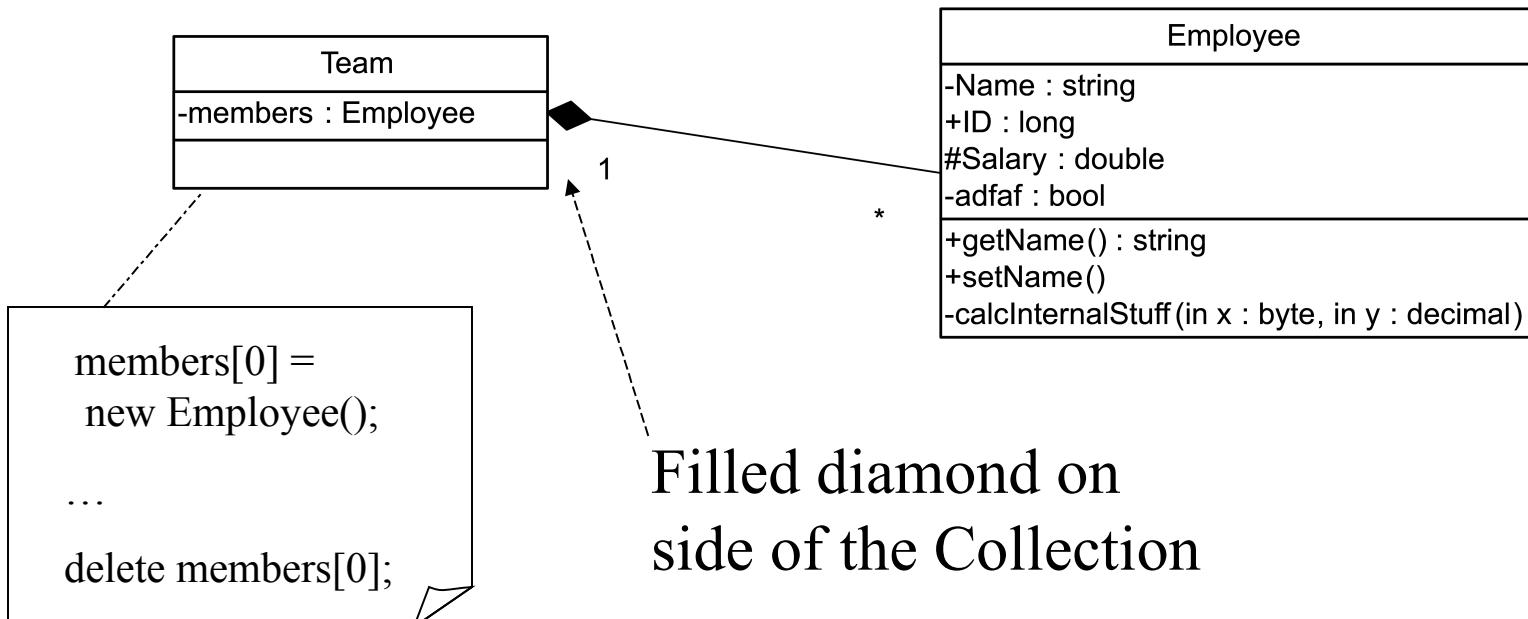
Hollow diamond on
the Collection side

No sole ownership implied

Composition

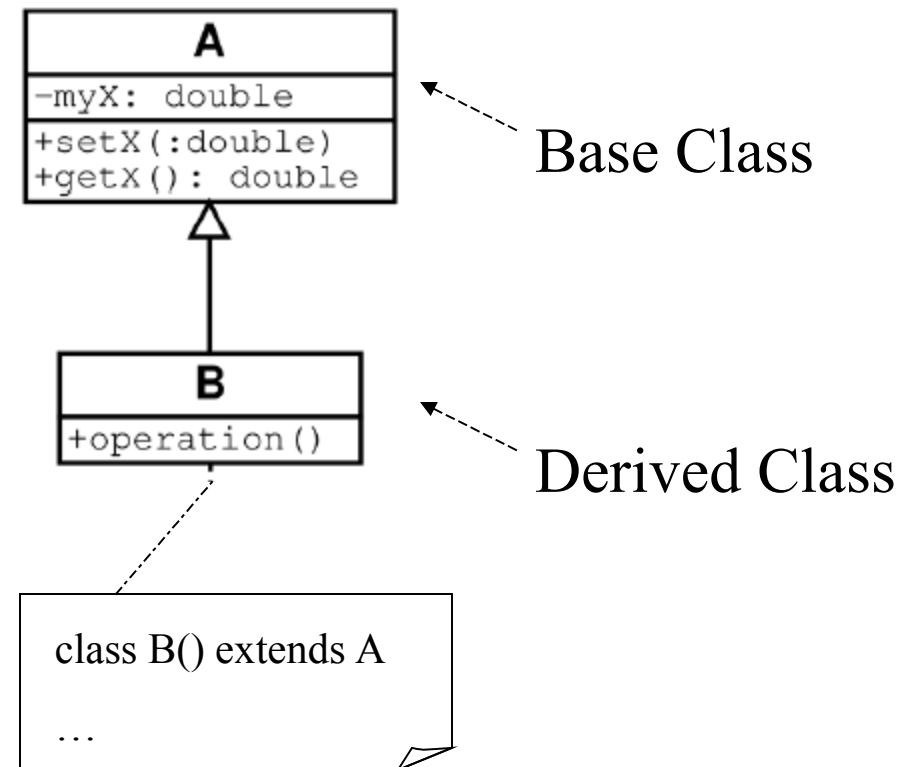
Composition is Aggregation with:

- Lifetime Control (owner controls construction, destruction)
- Part object may belong to only one whole object



Inheritance

Standard concept of inheritance

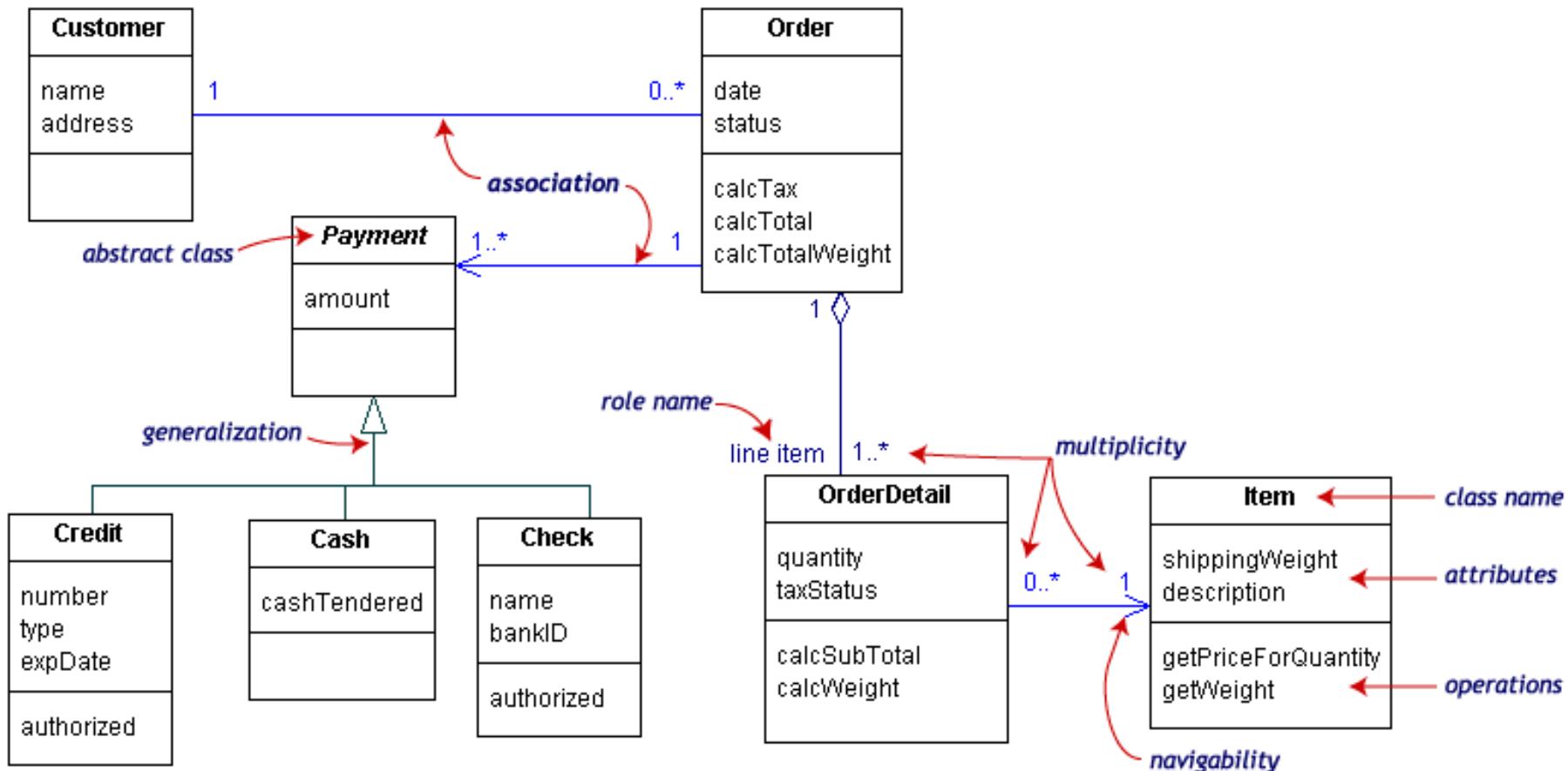


UML Multiplicities

Links on associations to specify more details about the relationship

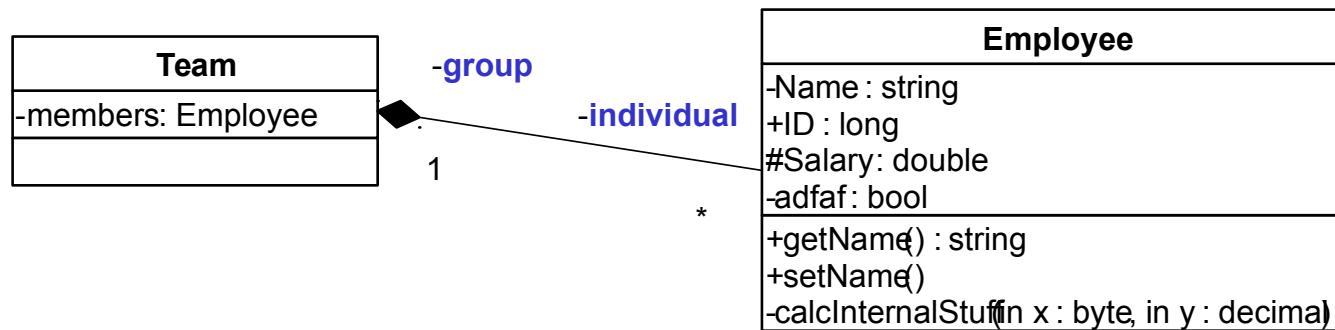
Multiplicities	Meaning
0..1	zero or one instance. The notation $n \dots m$ indicates n to m instances.
0..* or *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

UML Class Example



Association Details

- Can assign names to the ends of the association to give further information



Static vs. Dynamic Design

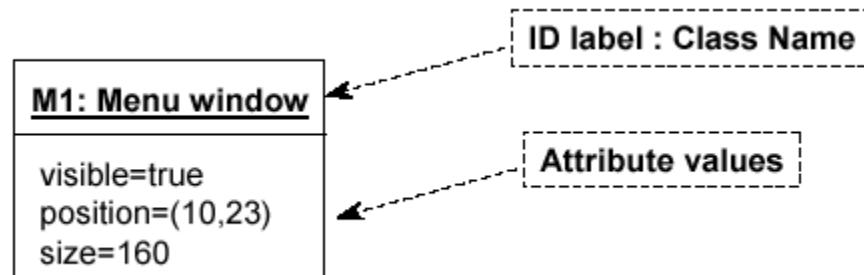
- Static design describes code structure and object relations
 - Class relations
 - Objects at design time
 - Doesn't change
- Dynamic design shows communication between objects
 - Similarity to class relations
 - Can follow sequences of events
 - May change depending upon execution scenario
 - Called Object Diagrams

Object Diagrams

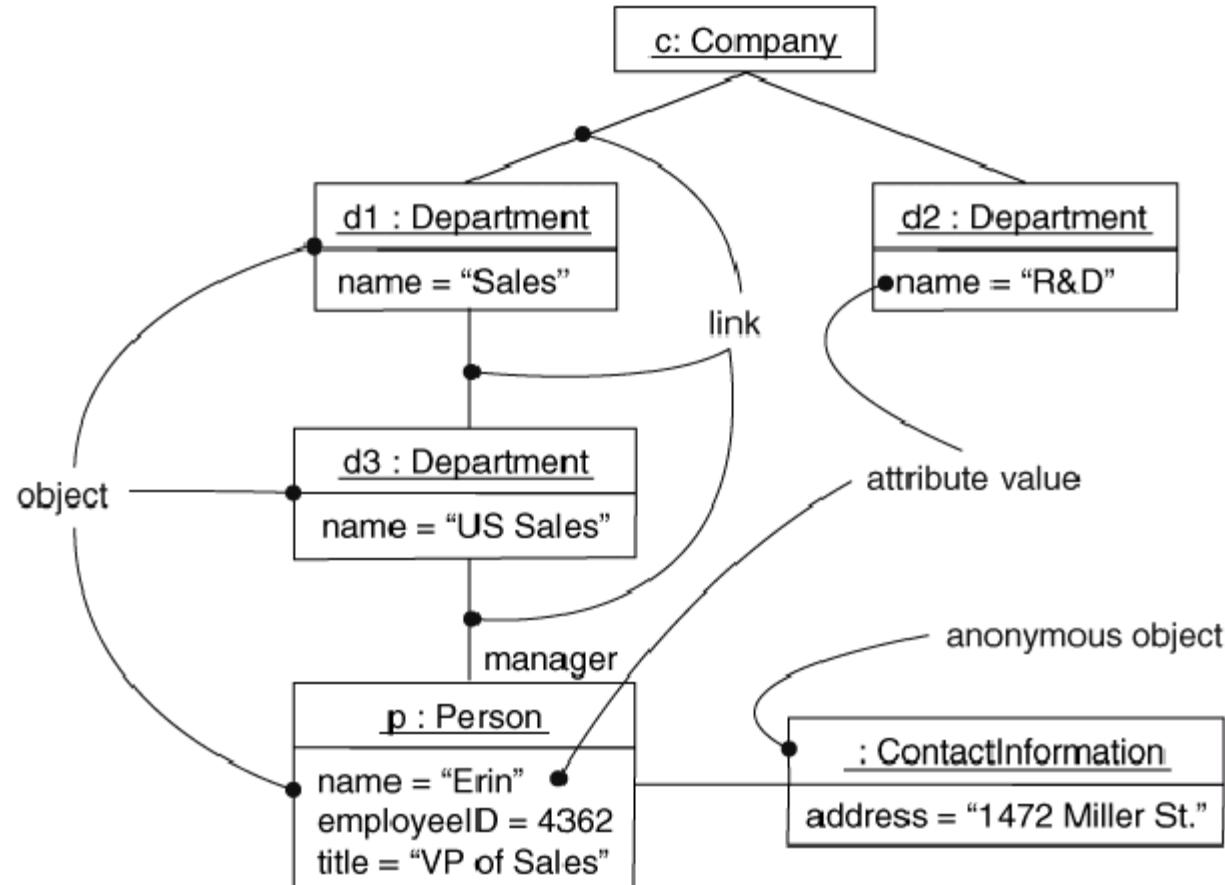
- Shows instances of Class Diagrams and links among them
 - An object diagram is a snapshot of the objects in a system
 - At a point in time
 - With a selected focus
 - Interactions – Sequence diagram
 - Message passing – Collaboration diagram
 - Operation – Deployment diagram

Object Diagrams

- Format is
 - Instance name : Class name
 - Attributes and Values
 - Example:



Objects and Links

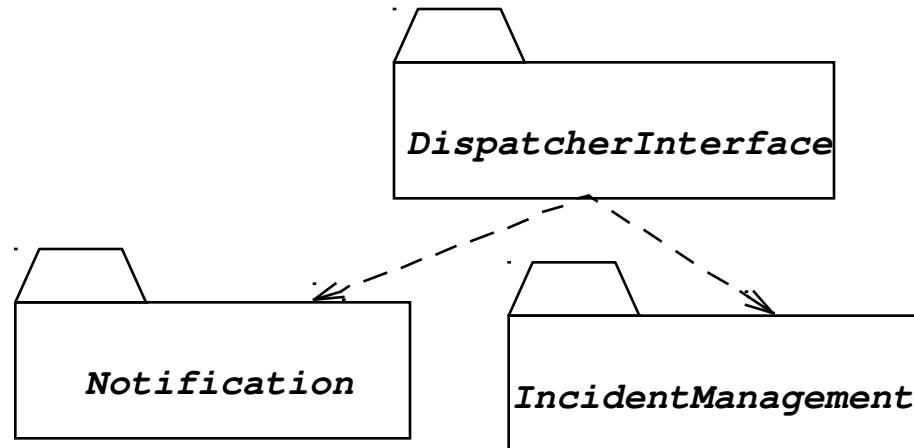


Can add association type and also message type

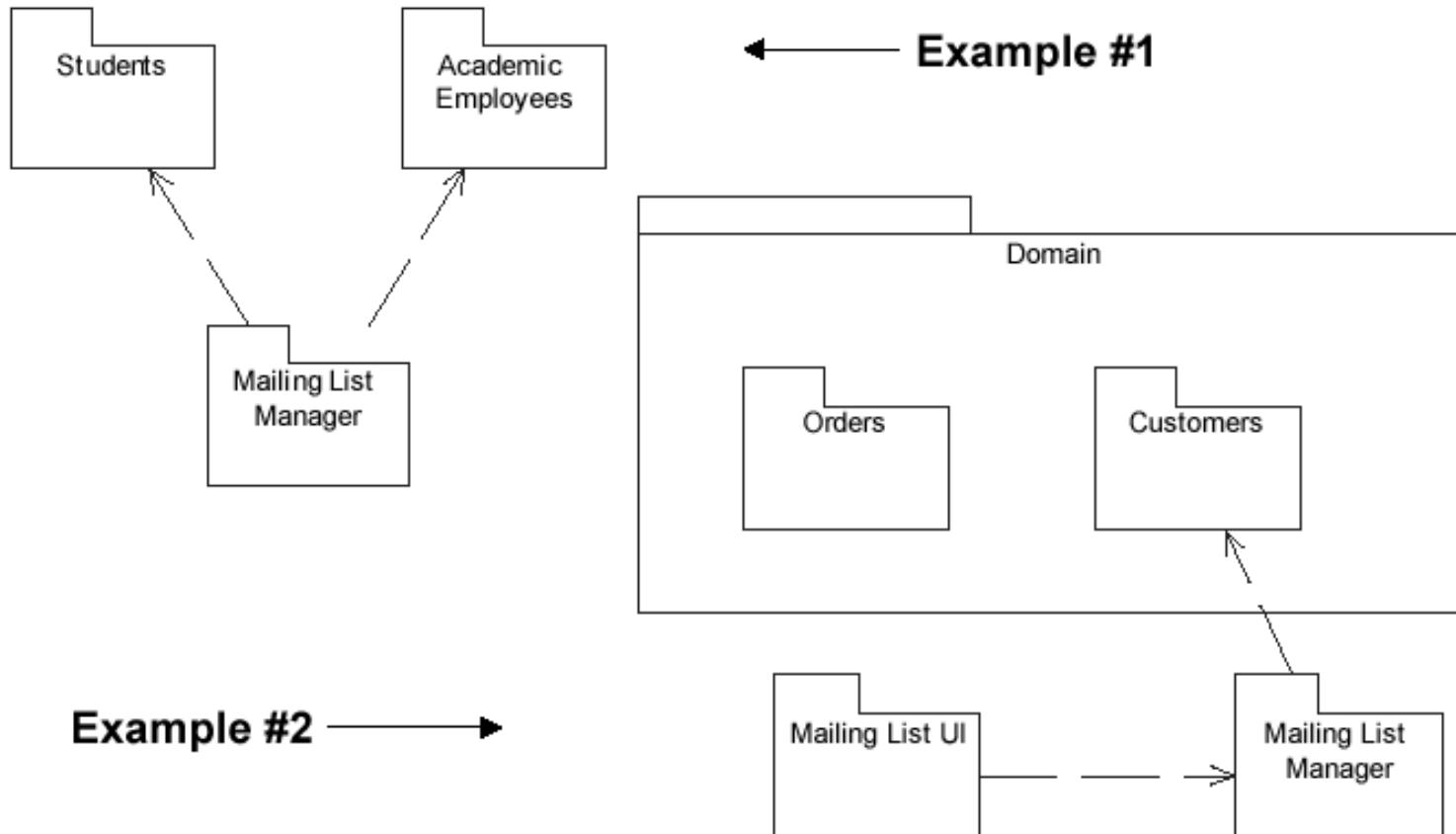
Package Diagrams

- To organize complex class diagrams, you can group classes into packages. A package is a collection of logically related UML elements
- Notation
 - Packages appear as rectangles with small tabs at the top.
 - The package name is on the tab or inside the rectangle.
 - The dotted arrows are dependencies. One package depends on another if changes in the other could possibly force changes in the first.
 - Packages are the basic grouping construct with which you may organize UML models to increase their readability

Package Example



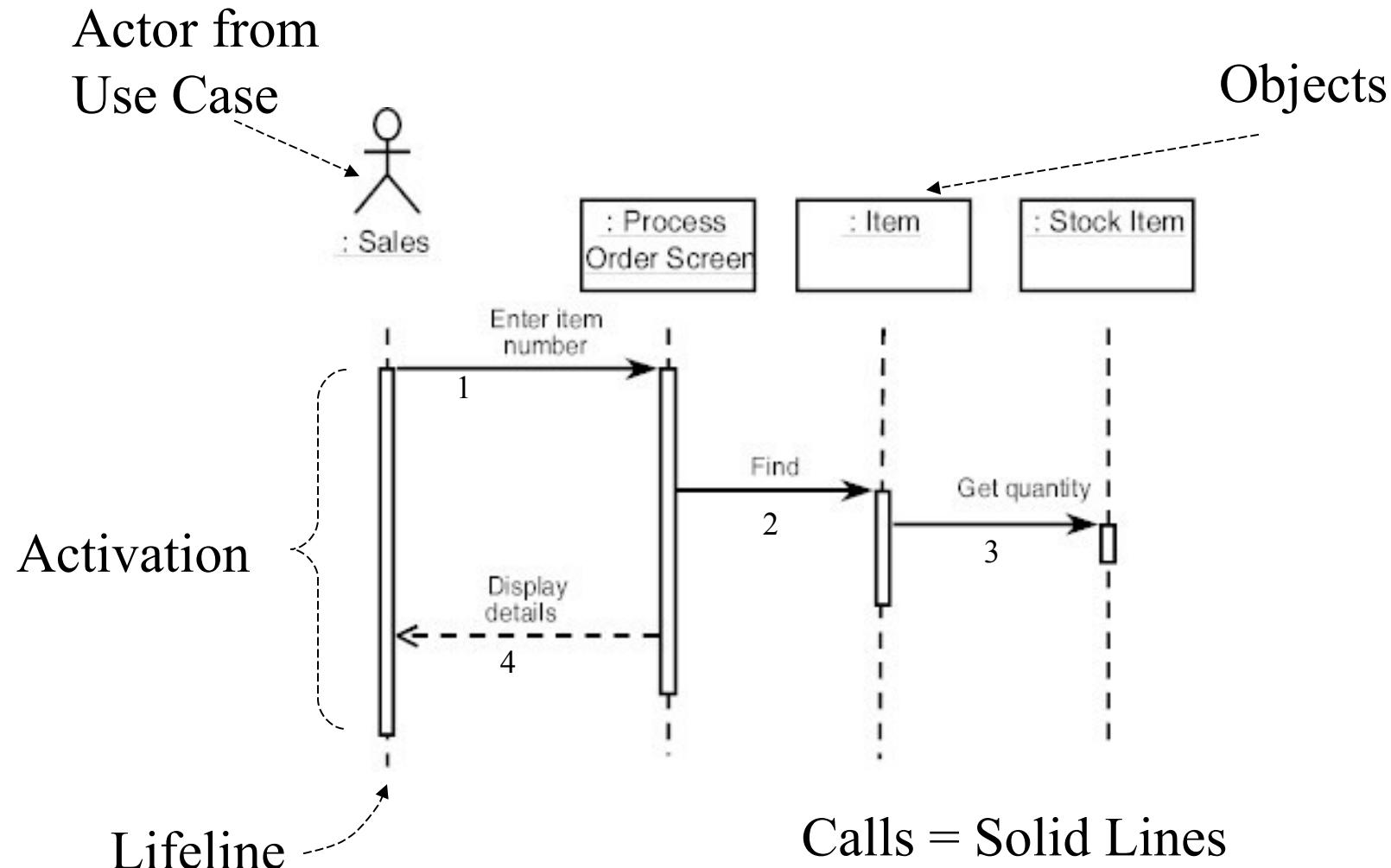
More Package Examples



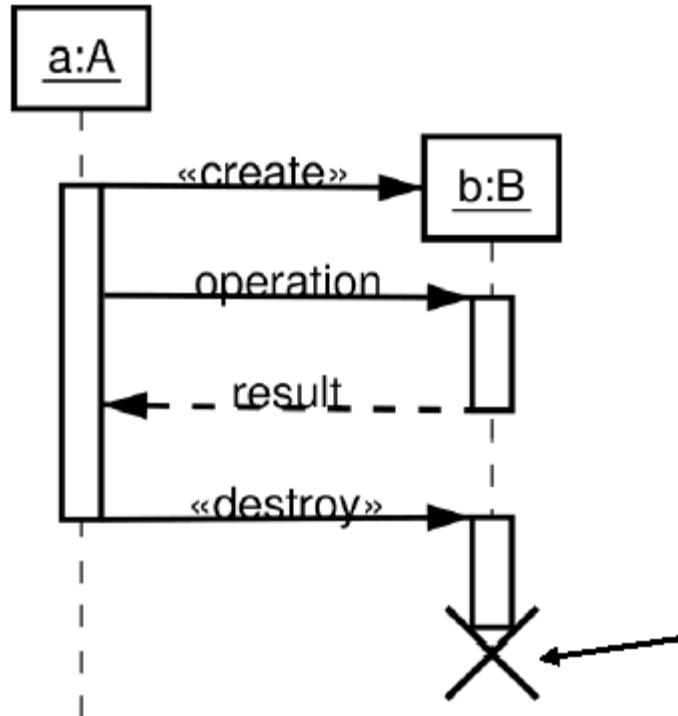
Interaction Diagrams

- Interaction diagrams are dynamic -- they describe how objects collaborate.
- A Sequence Diagram:
 - Indicates what messages are sent and when
 - Time progresses from top to bottom
 - Objects involved are listed left to right
 - Messages are sent left to right between objects in sequence

Sequence Diagram Format



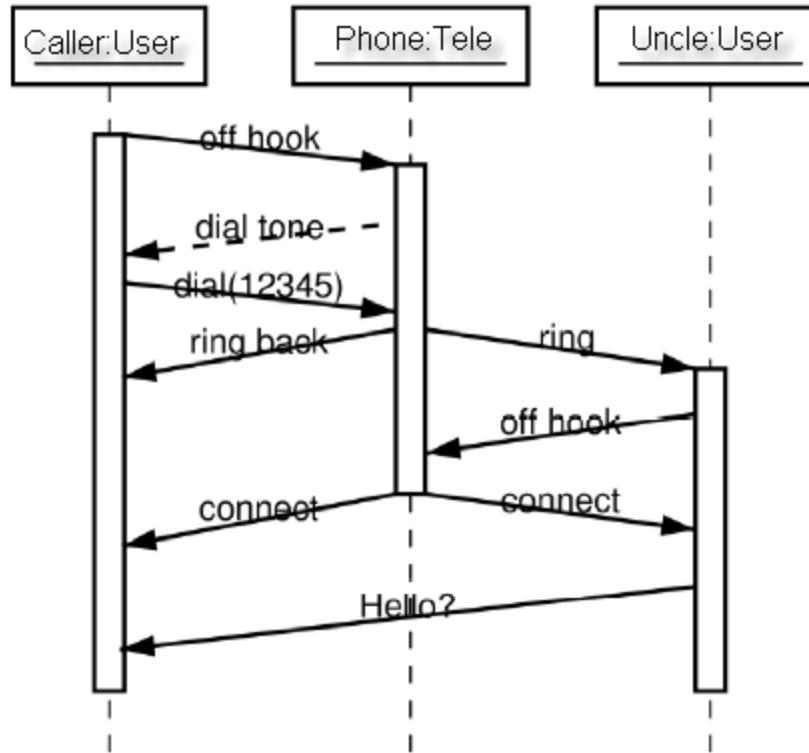
Sequence Diagram: Destruction



Shows Destruction of b
(and Construction)

Sequence Diagram: Timing

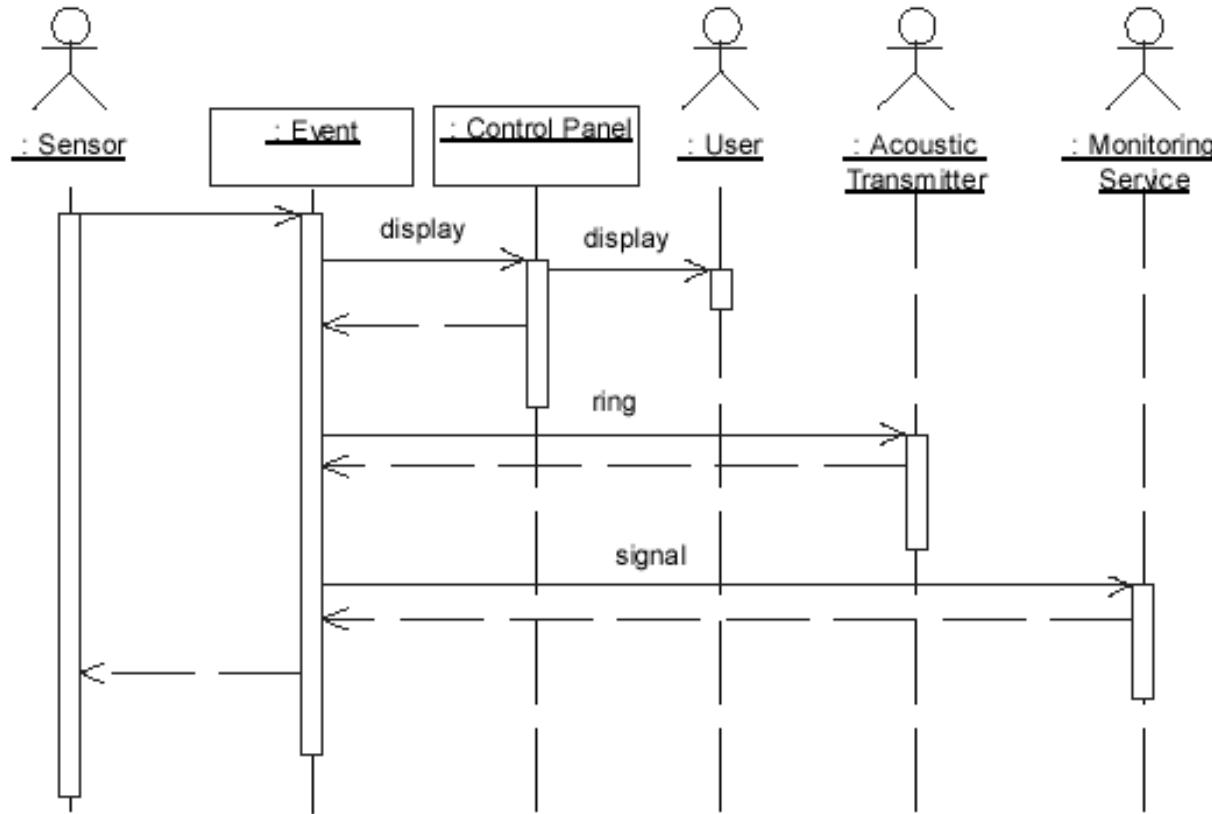
Slanted Lines show propagation delay of messages
Good for modeling real-time systems



If messages cross this is usually problematic – race conditions

Sequence Example: Alarm System

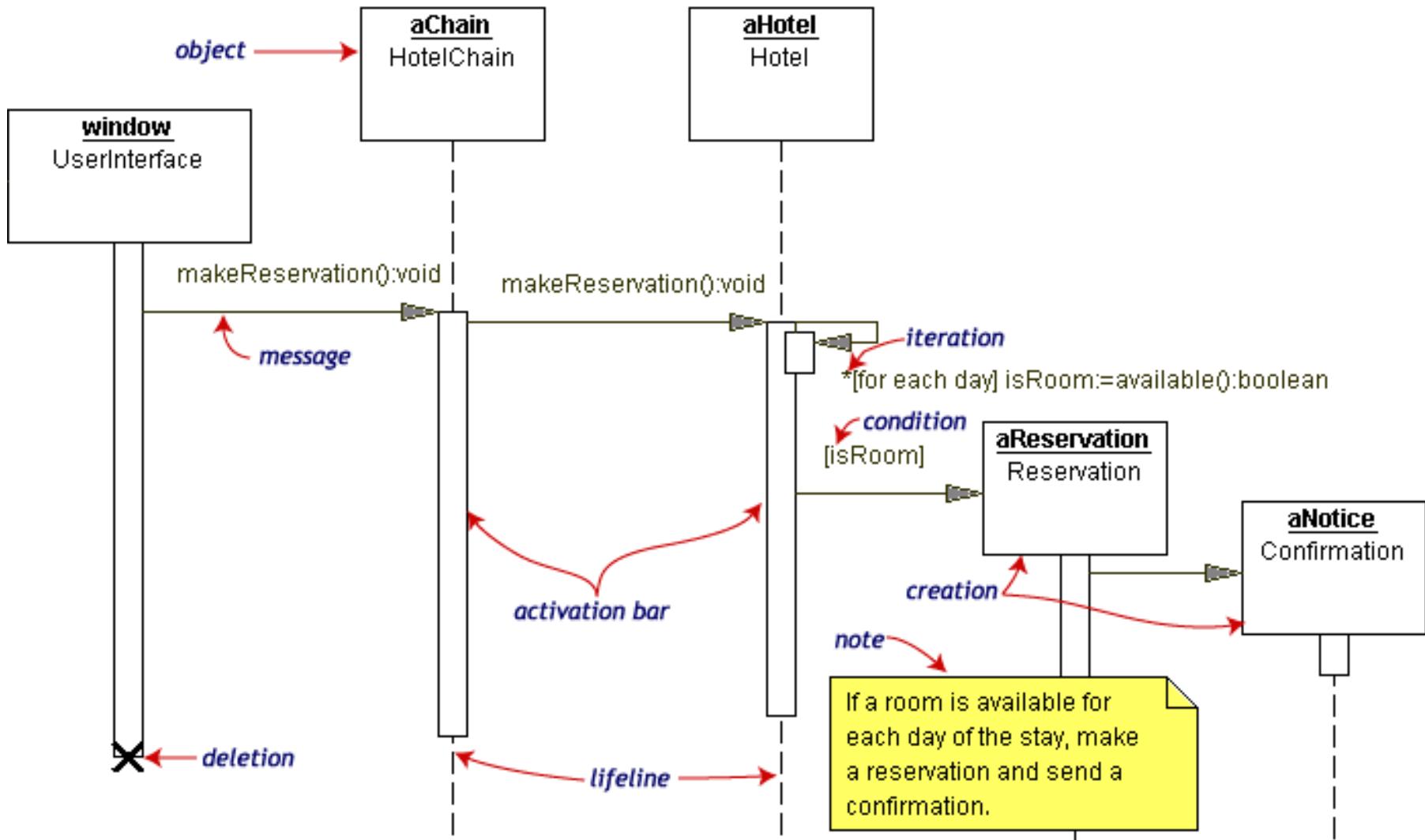
- When the alarm goes off, it rings the alarm, puts a message on the display, notifies the monitoring service



Sequence Diagram:

Example

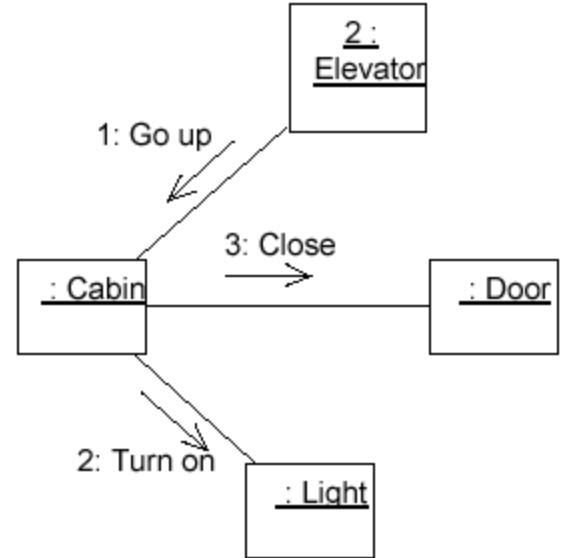
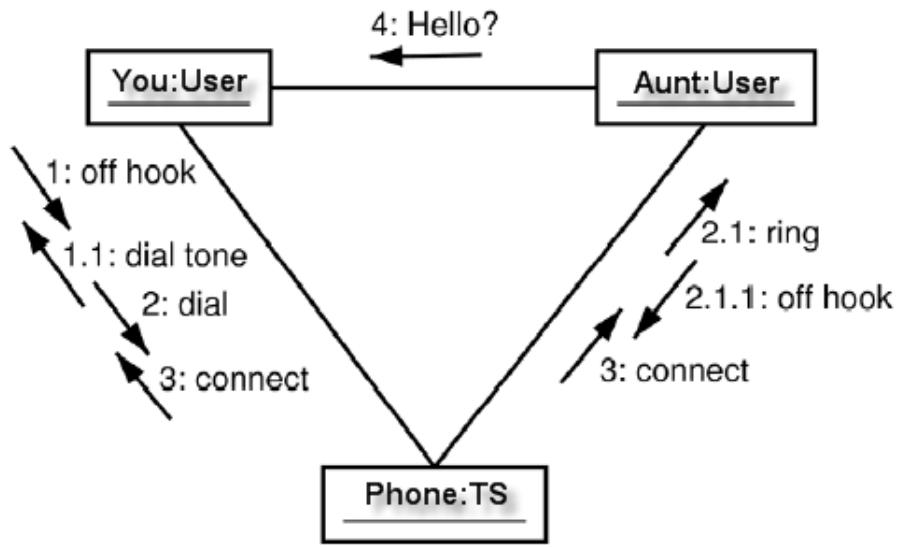
Hotel Reservation



Collaboration Diagram

- Collaboration Diagrams show similar information to sequence diagrams, except that the vertical sequence is missing. In its place are:
 - Object Links - solid lines between the objects that interact
 - On the links are Messages - arrows with one or more message name that show the direction and names of the messages sent between objects
- Emphasis on static links as opposed to sequence in the sequence diagram

Collaboration Diagram



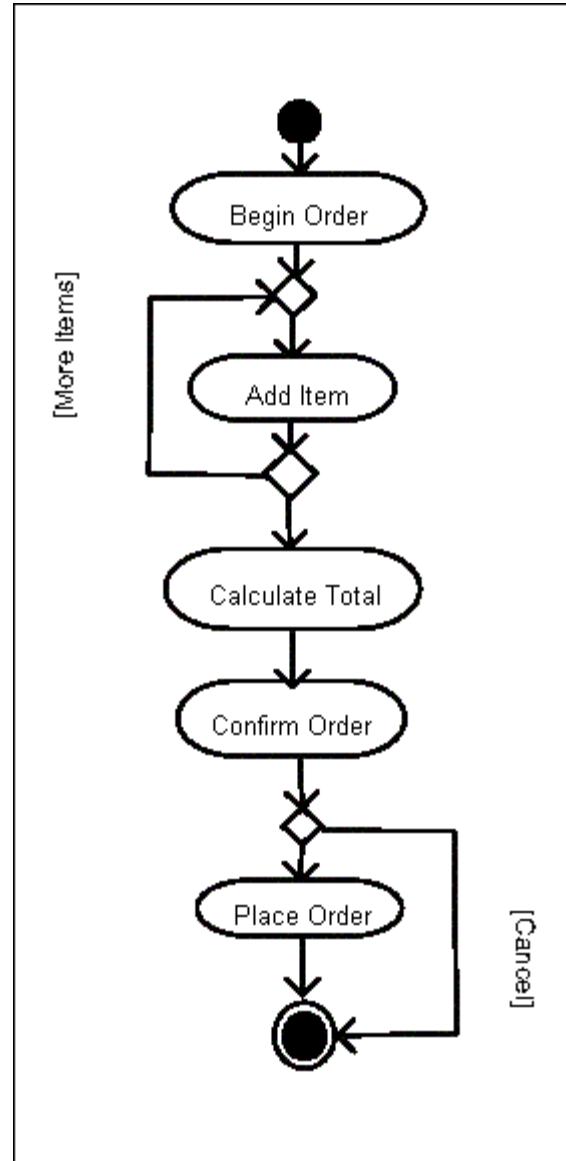
Activity Diagrams

- Fancy flowchart
 - Displays the flow of activities involved in a single process
 - States
 - Describe what is being processed
 - Indicated by boxes with rounded corners
 - Swim lanes
 - Indicates which object is responsible for what activity
 - Branch
 - Transition that branch
 - Indicated by a diamond
 - Fork
 - Transition forking into parallel activities
 - Indicated by solid bars
 - Start and End

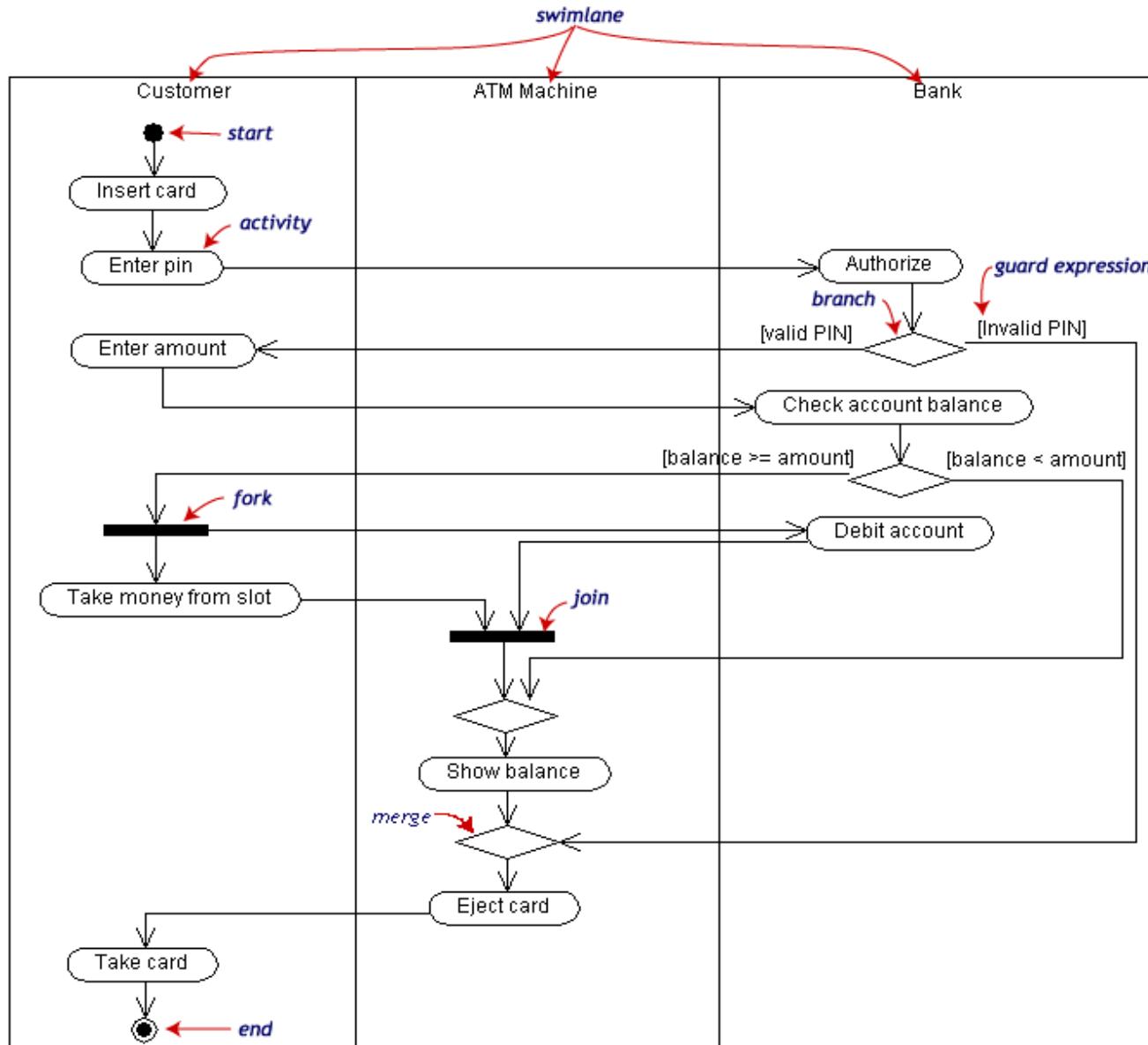


Sample Activity Diagram

- Ordering System
- May need multiple diagrams from other points of view



Activity Diagram: Example



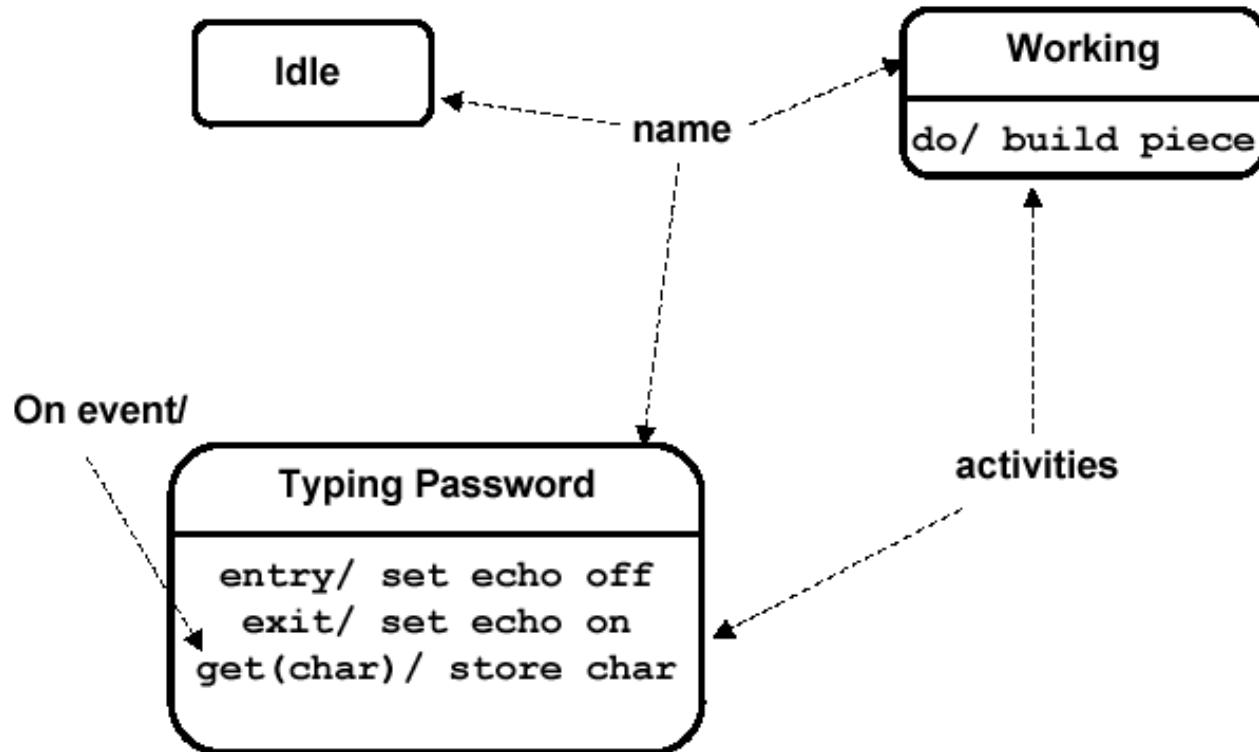
State Transition Diagrams

- Fancy version of a DFA
- Shows the possible states of the object and the transitions that cause a change in state
 - i.e. how incoming calls change the state
- Notation
 - States are rounded rectangles
 - Transitions are arrows from one state to another. Events or conditions that trigger transitions are written beside the arrows.
 - Initial and Final States indicated by circles as in the Activity Diagram
 - Final state terminates the action; may have multiple final states

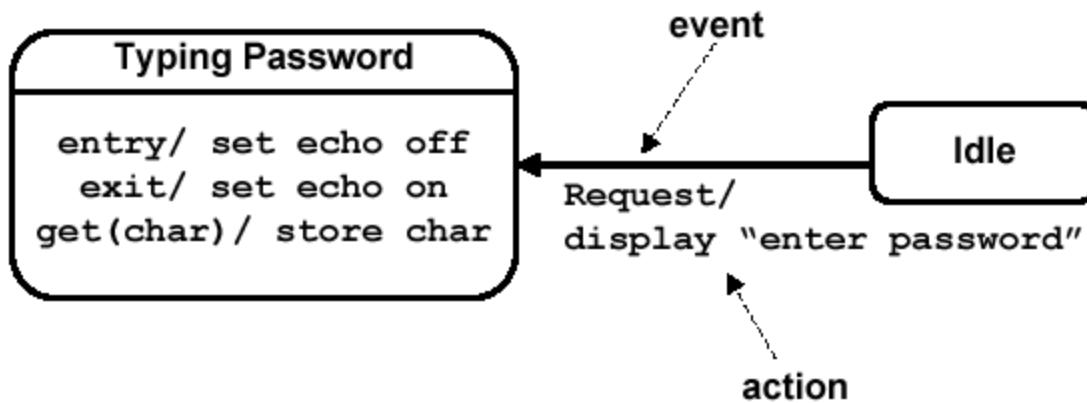
State Representation

- The set of properties and values describing the object in a well defined instant are characterized by
 - Name
 - Activities (executed inside the state)
 - Do/ activity
 - Actions (executed at state entry or exit)
 - Entry/ action
 - Exit/ action
 - Actions executed due to an event
 - Event [Condition] / Action ^Send Event

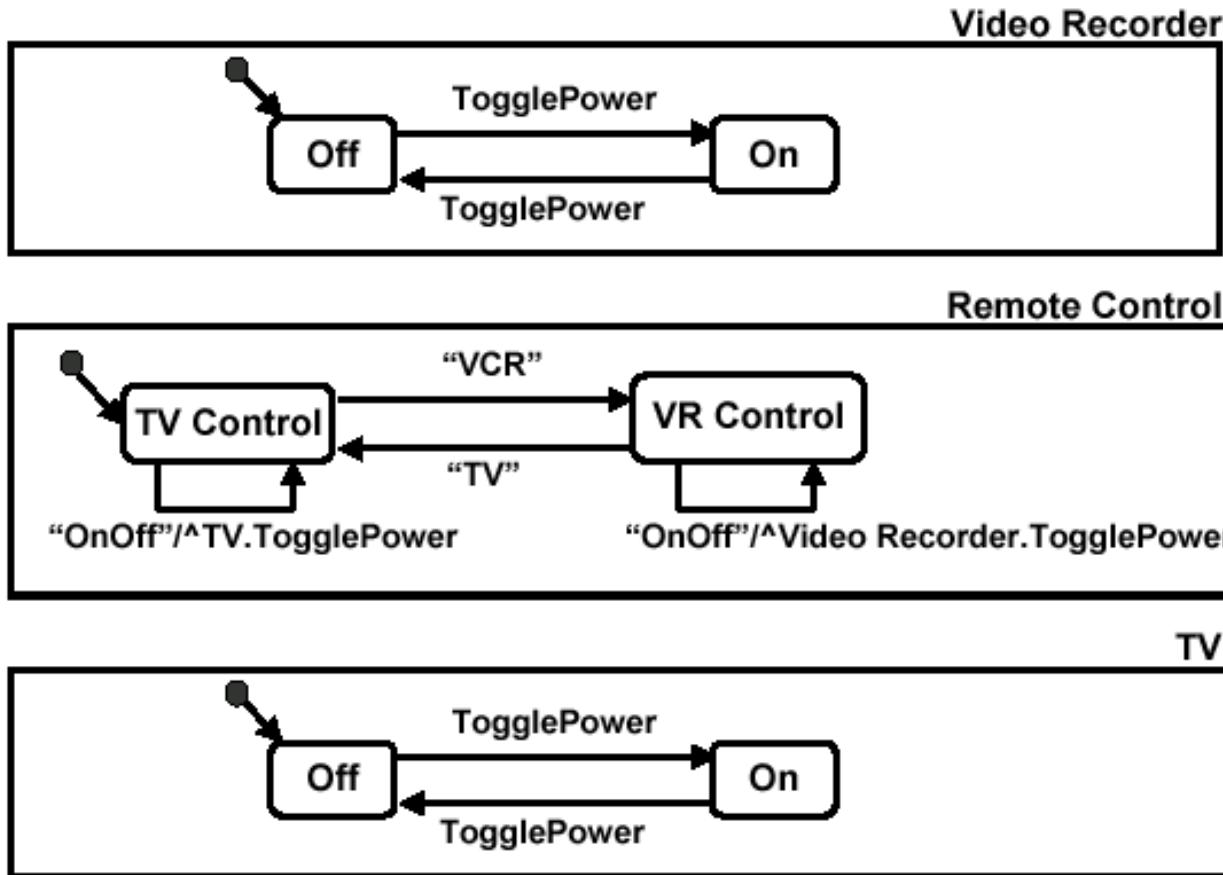
Notation for States



Simple Transition Example

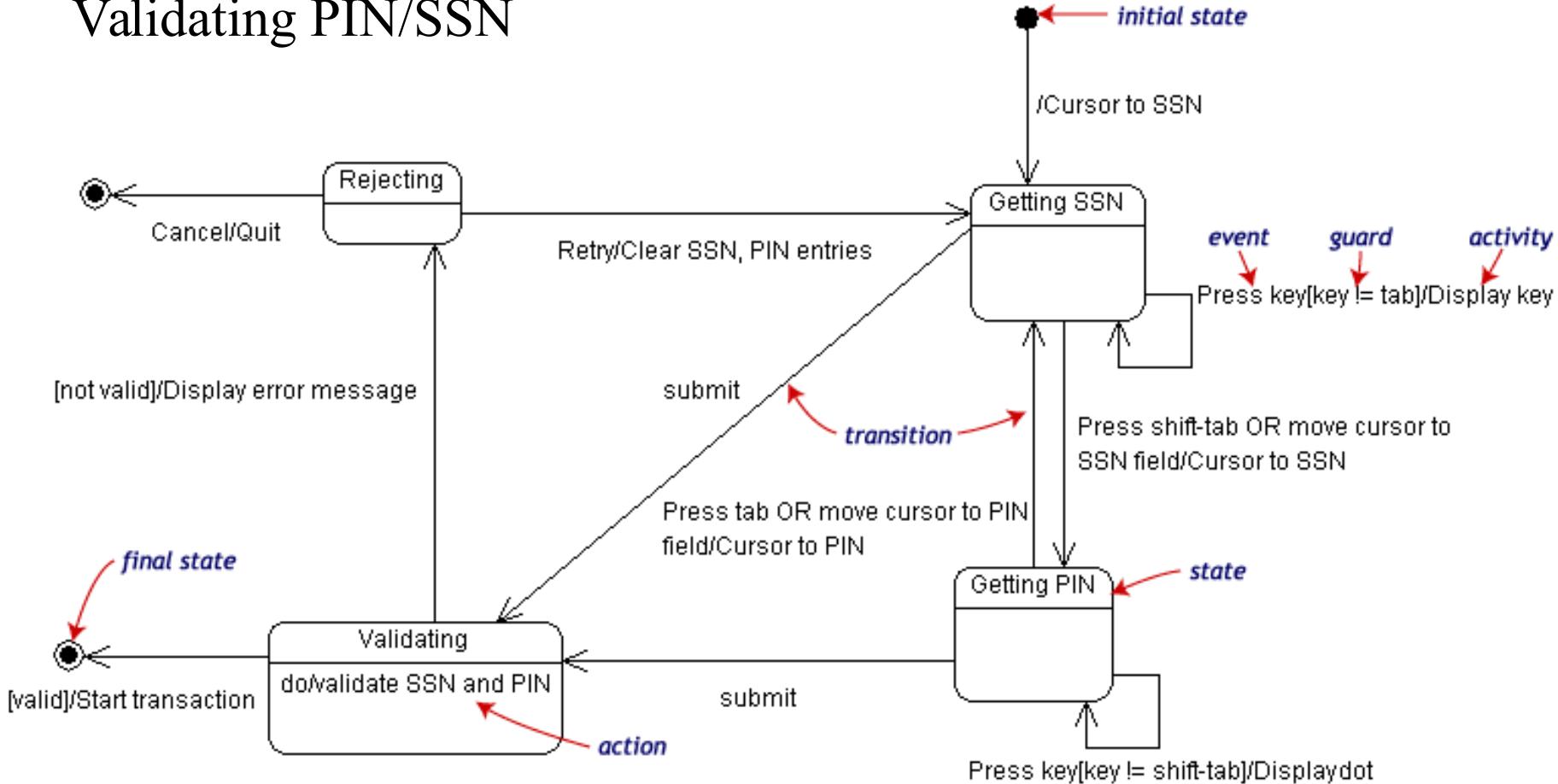


Simple State Examples...



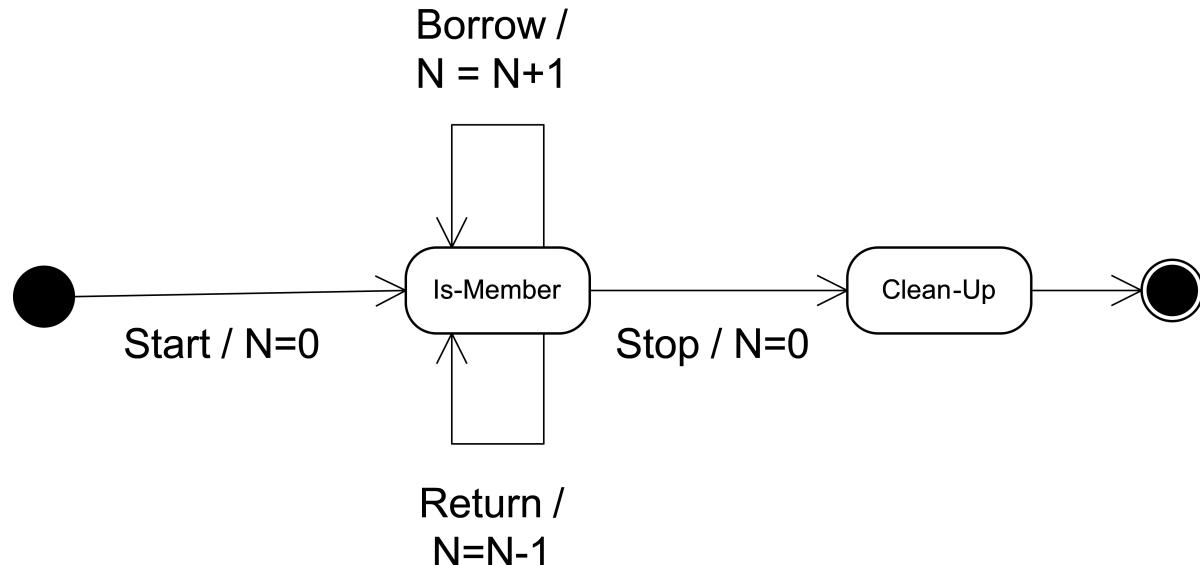
State Transition Example

Validating PIN/SSN



State Charts: Local Variables

- State Diagrams can also store their own local variables, do processing on them
- Library example counting books checked out and returned

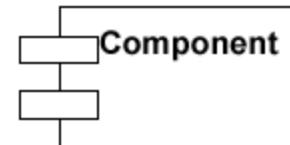


Component Diagrams

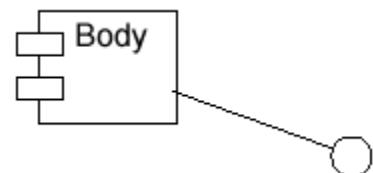
- Shows various components in a system and their dependencies, interfaces
- Explains the structure of a system
- Usually a physical collection of classes
 - Similar to a Package Diagram in that both are used to group elements into logical structures
 - With Component Diagrams all of the model elements are private with a public interface whereas Package diagrams only display public items.

Component Diagram Notation

- Components are shown as rectangles with two tabs at the upper left

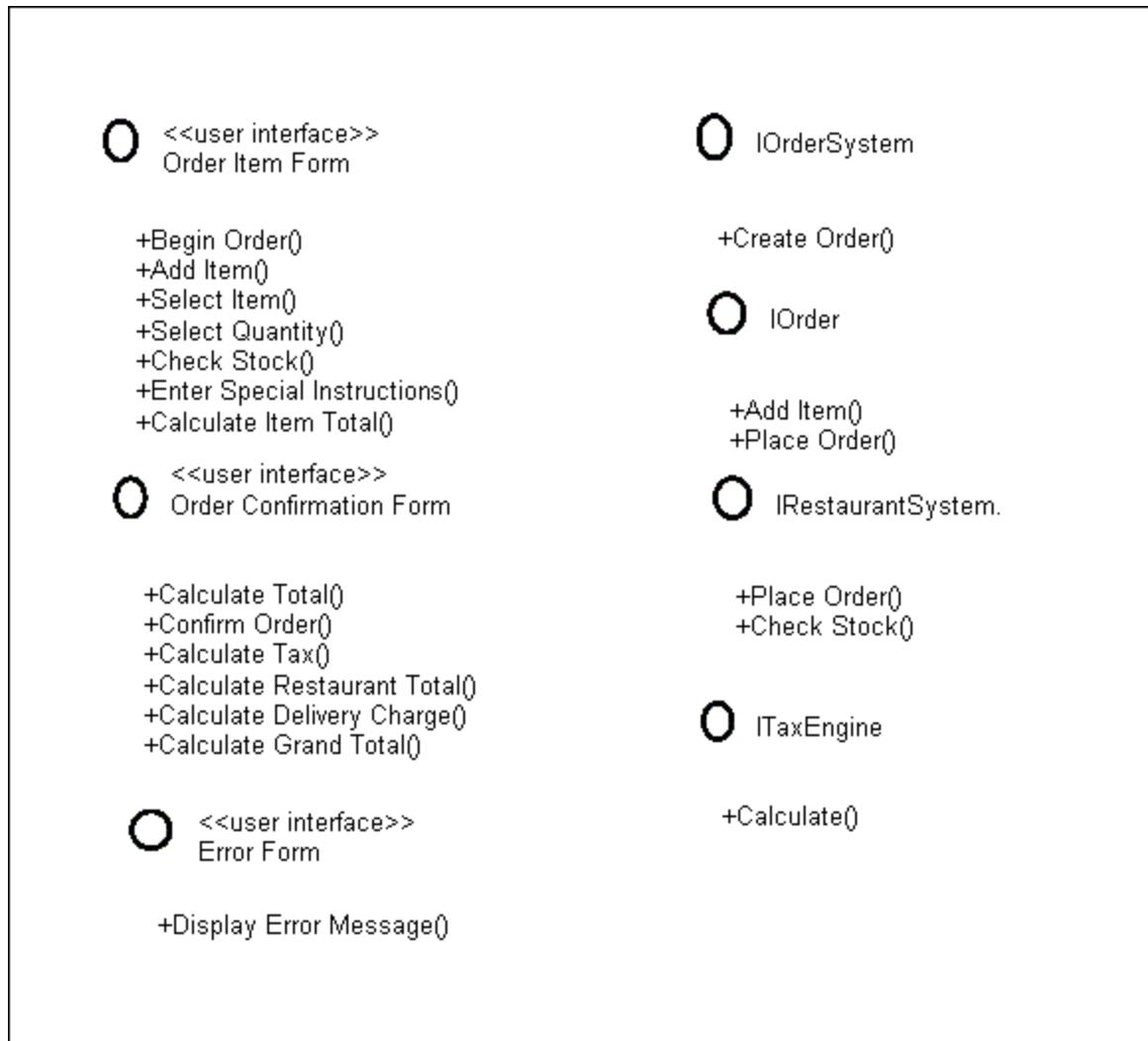


- Dashed arrows indicate dependencies
- Circle and solid line indicates an interface to the component



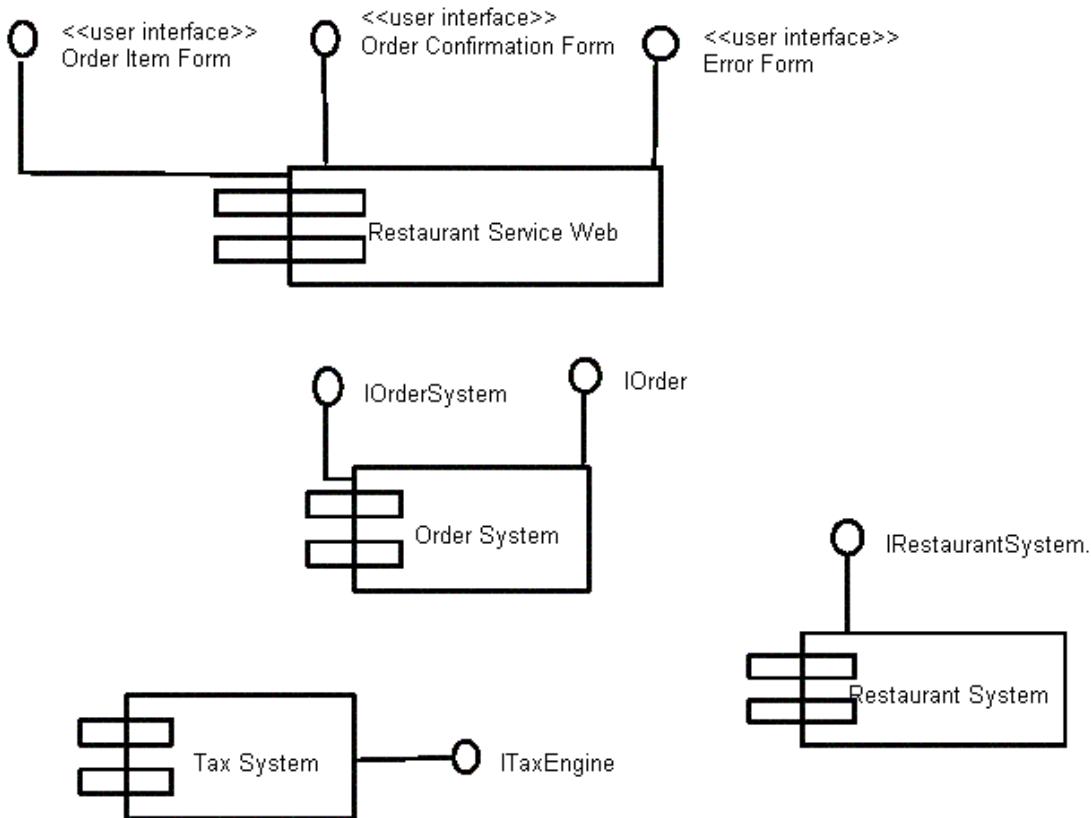
Component Example: Interfaces

- Restaurant ordering system
- Define interfaces first – comes from Class Diagrams



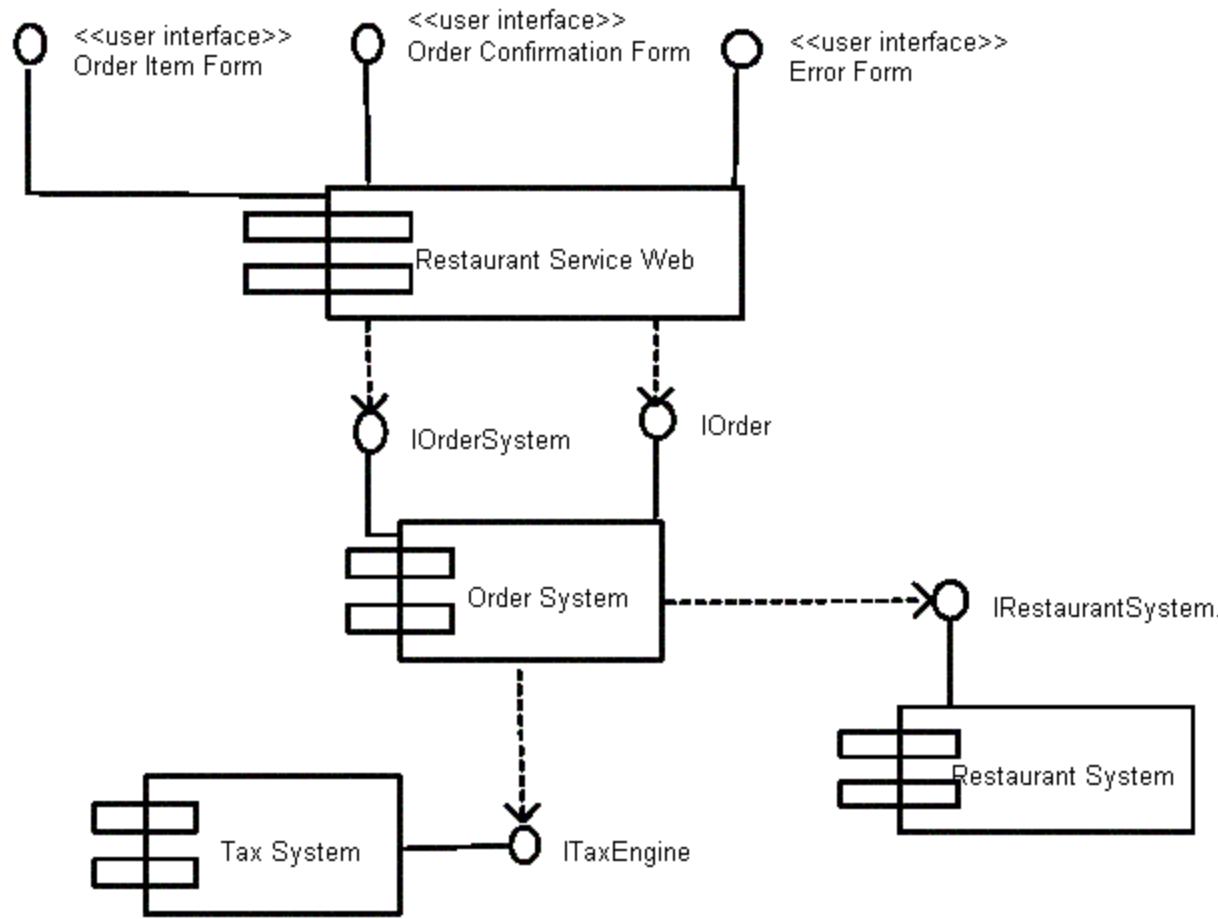
Component Example: Components

- Graphical depiction of components



Component Example: Linking

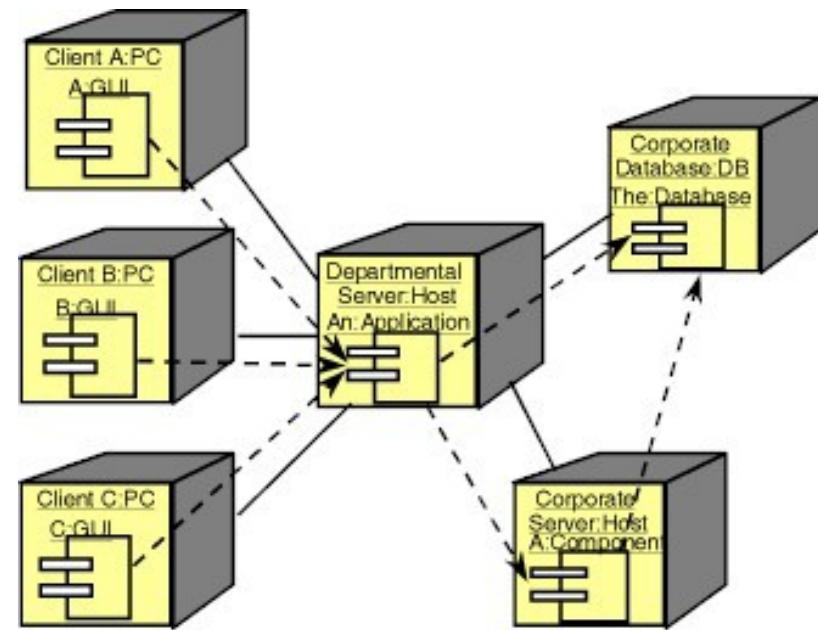
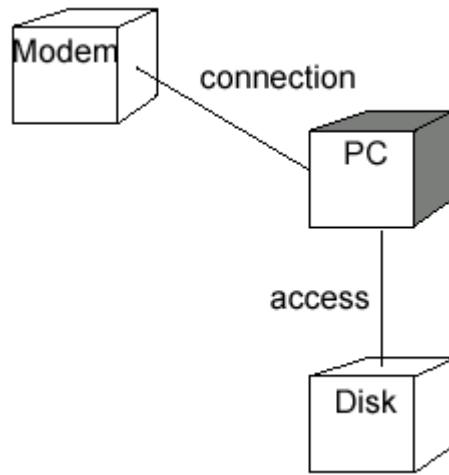
- Linking components with dependencies



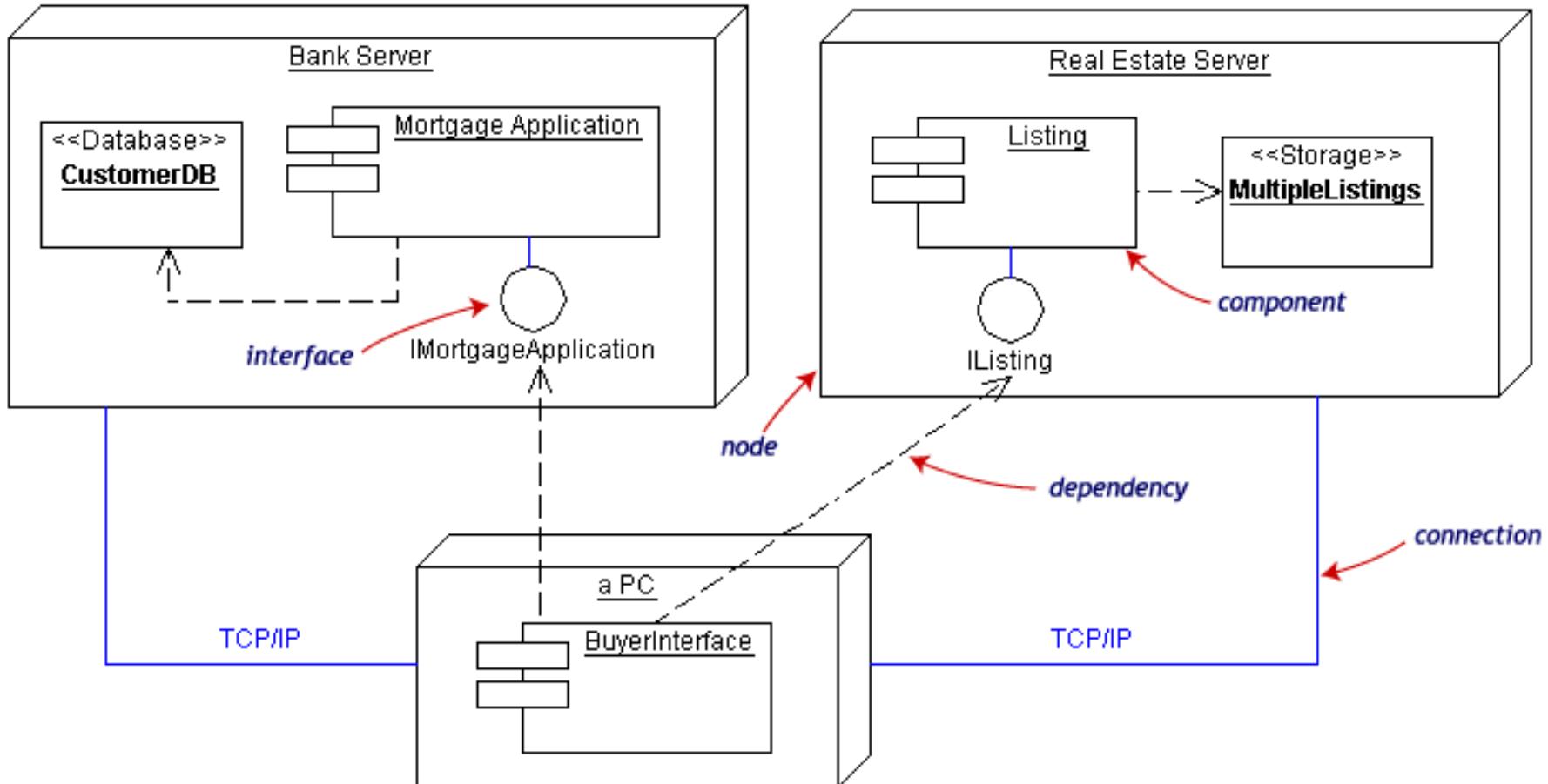
Deployment Diagrams

- Shows the physical architecture of the hardware and software of the deployed system
- Nodes
 - Typically contain components or packages
 - Usually some kind of computational unit; e.g. machine or device (physical or logical)
- Physical relationships among software and hardware in a delivered systems
 - Explains how a system interacts with the external environment

Deployment Examples



Deployment Example



Often the Component Diagram is combined with the Deployment

Summary and Tools

- UML is a modeling language that can be used independent of development
- Adopted by OMG and notation of choice for visual modeling
 - <http://www.omg.org/uml/>
- Creating and modifying UML diagrams can be labor and time intensive.
- Lots of tools exist to help
 - Tools help keep diagrams, code in sync
 - Repository for a complete software development project
 - Examples tools Microsoft Visio, Dia

Thank You

Object-oriented programming - relations

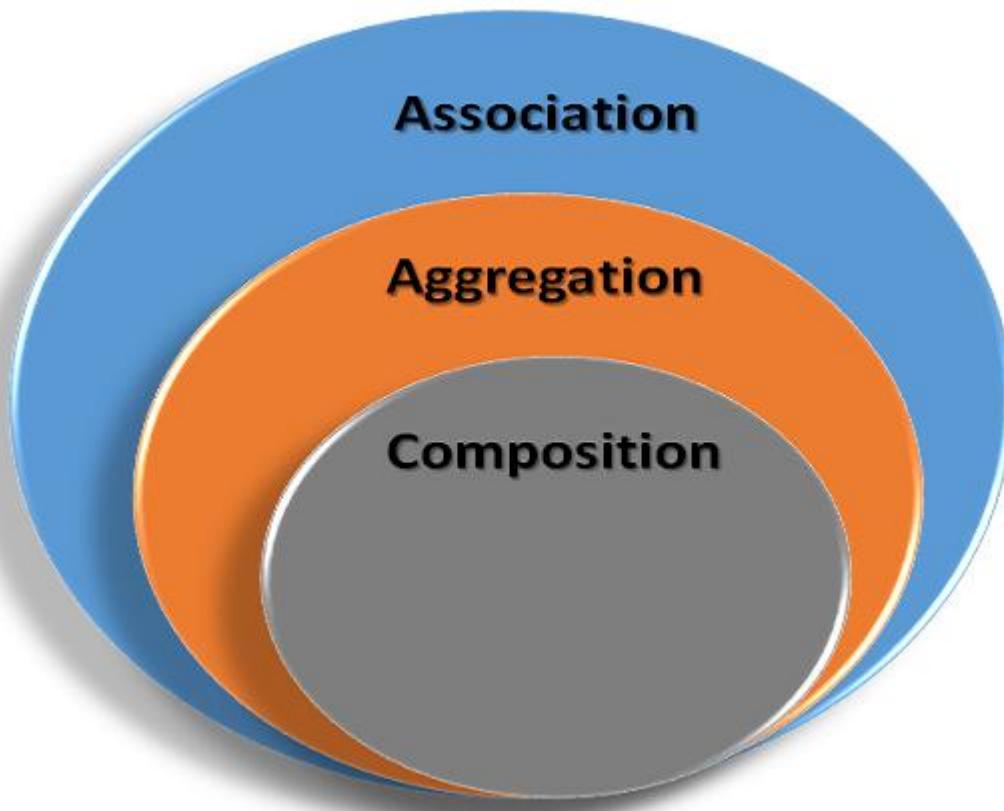
Rafał Witkowski

2021

Relations

- Code reuse is one of the many benefits of OOP (object-oriented programming). Reusability is feasible because of the various types of relationships that can be implemented among classes. This presentation will demonstrate the types of relationships (from weak to strong) using Java code samples and the symbols in the UML (unified modeling language) class diagram.

Relations



Dependency (Using)

- Is a relationship when objects of a class work briefly with objects of another class. Normally, multiplicity doesn't make sense on a dependency



Dependency - Java

```
class Move {  
    public void Roll() { ... }  
}
```

```
class Player {  
    public void PerformAction(Move move) {  
        move.Roll();  
    }  
}
```

Association

- This relationship transpires when objects of one class know the objects of another class. The relationship can be one to one, one to many, many to one, or many to many. Moreover, objects can be created or deleted independently.



Association – java

```
public static void main (String[] args) {  
    Doctor doctorObj = new Doctor("Rick");  
    Patient patientObj = new Patient("Morty");  
    System.out.println(patientObj.getPatientName() +  
        " is a patient of " + doctorObj.getDoctorName());  
}
```

Aggregation

- Is a "has-a" type relationship and is a one-way form of association. This relationship exists when a class owns but shares a reference to objects of another class.



Aggregation – java

```
class Address{  
//code here  
}
```

```
class StudentClass{  
    private Address studentAddress;  
//code here  
}
```

```
Public StudentClass(Address adres)  
{  
    studentAddress = adres;  
}
```

Composition

- Is a "part-of" type of relationship, and is a strong type of association. In other words, composition happens when a class owns & contains objects of another class



Composition – java

```
class Room {  
    //code here  
}
```

```
class House {  
    private Room room;  
    //code here  
}
```

```
Public House()  
{  
    room = new Room();  
}
```

Inheritance

- Is an "is-a" type of relationship. It's a mechanism that permits a class to acquire the properties and behavior of an existing class (or sometimes more classes, if the language allows multiple inheritance).



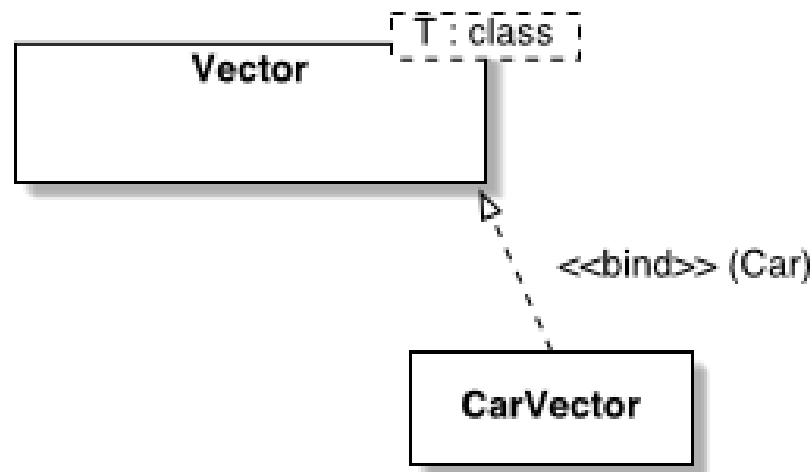
Inheritance – java

```
class Vehicle {  
//code here  
}
```

```
class Car extends Vehicle {  
//code here  
}
```

Instantiation (generics)

- A generic class arises from a situation where the behavior of a class can be abstracted into something which is relevant to more than one type of state. The goal is to share the definition (in the class) of some behavior (i.e. the methods) across different data types.



Polymorphism

- For OO languages polymorphism is tied up with substitutability. We design methods and we write client code that can operate on a set of types. What is common about these types is that they are substitutable for each other.
- The idea that the code that is executed as the result of a message being sent depends on the class of the object that receives the message. Objects of different classes can react differently to being sent the same method in a message.

Object-oriented programming

Rafał Witkowski

2021



Mikołaj K [REDACTED]

Poza tym nie wiem czy słyszeliście państwo o takiej technice jak programowanie obiektowe - chodzi z grubsza o to, że obiekty którymi otacza się ludzi mają wpływ na nasze mózgi tak żebyśmy byli zniewoleni.

Programowanie obiektowe nawet wykorzystuje swoją specjalną technikę która nazywa się Jawa - przez to ludzie myślą że simulacja na która patrzą to tak naprawdę java czyli jest prawdziwa

JOE MONSTER

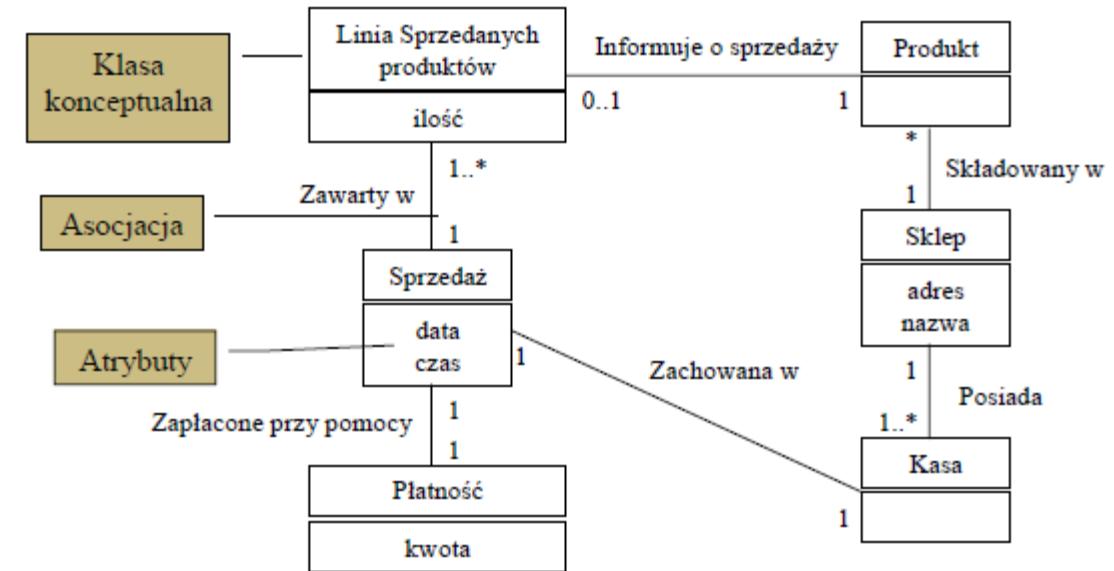
Przed chwilą Lubię to! Więcej

Domain model

- The domain model aims to visualization of concepts (conceptual classes) occurring in the field in question (the context of the system being set up).
- The most important model is created during object-oriented analysis! Work need to be done at this stage is to identify the conceptual classes. Correctness the construction of this model guarantees success during the design and implementation phase.
- The disciplinary knowledge model is a visual representation of classes or real objects occurring in the real environment that we model. It is not a representation of programming entities, i.e. classes and objects written in the language programming.

Domain model – example

- In UML, the domain knowledge model is written using the class diagram it describes:
 - Real objects from a given domain or conceptual classes,
 - Associations between conceptual classes,
 - Attributes of conceptual classes.



Domain model

- The domain knowledge model visualizes and combines conceptual classes occurring in a particular field.
- It refers to certain abstractions of conceptual classes because a given concept may represent different concepts depending on the context in question.
- Alternatively, concepts written by means of UML notation could be written using text in natural language, or a dictionary. However, visual language makes it easier to understand the concepts described and is an excellent way of communication knowledge about the created system among the team members.
- The domain model of knowledge in a given field can be treated as a visual dictionary of the relevant elements of the field under consideration.

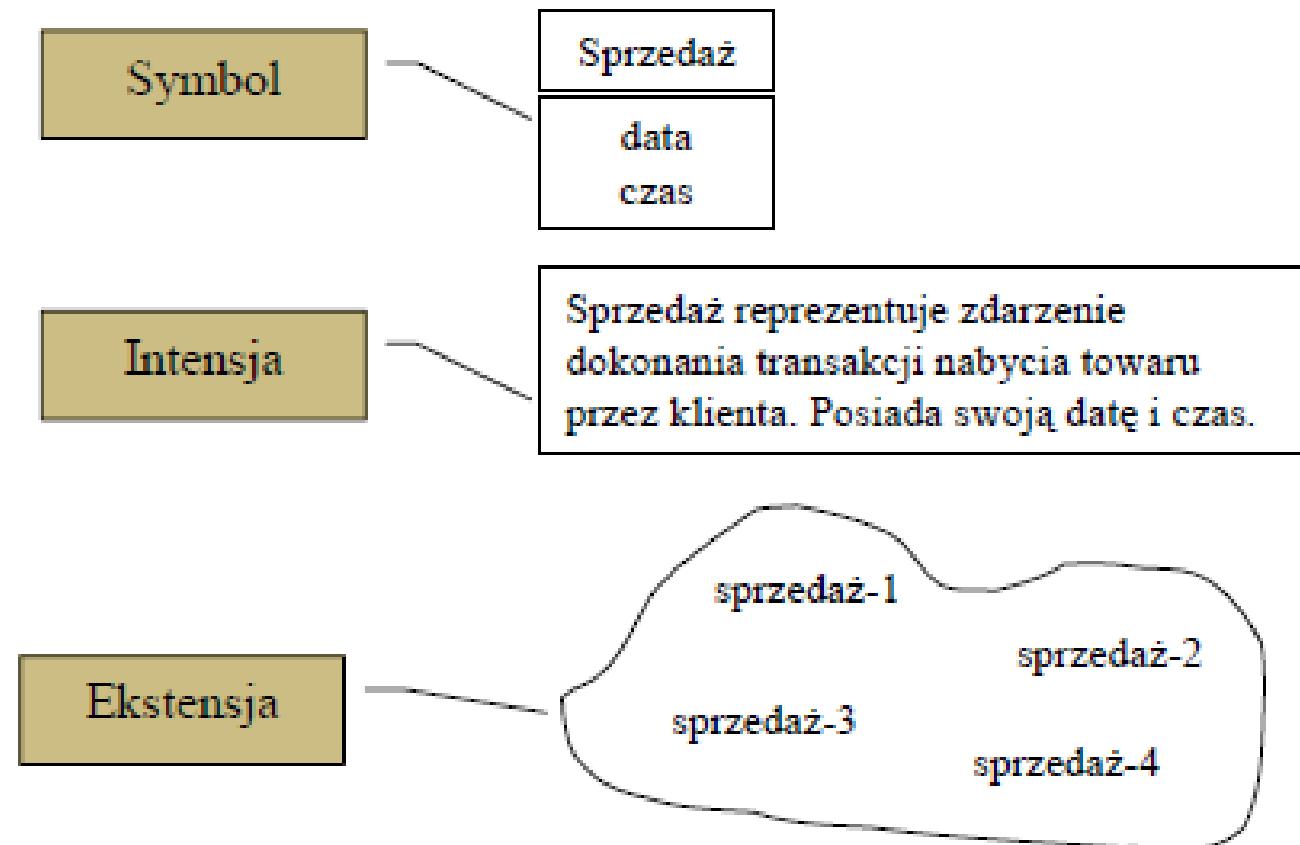
Domain model

- The domain knowledge model visualizes things from the real world, not classes written in programming languages such as C++ or Java.
- In models of this type you should avoid:
 - Programming entities such as the Window or Database,
 - Responsibility of objects and their methods.

Doamin model – creation process

- A conceptual class is an idea, a thing or an object.
- A conceptual class can be considered in three ways:
 - Symbol - a word or graphic element representing a conceptual class,
 - Intensity - definition of a conceptual class,
 - Extension - a set of examples to which a given conceptual class applies.

Domain model – creation



Identification of conceptual classes

- In an iterative process, the domain model of knowledge is not built in one step, but in stages corresponding to the scope of the work of each iteration.
- The identification is carried out on the basis of the results of the requirements phase (e.g. on the basis of use cases).
- It is assumed that for the identification of conceptual classes it is good to go beyond the specification of the knowledge model by the introduction of many (sometimes too many) very detailed measures conceptual classes rather than unspecifying the model.
- It is common to omit (because they are not found) in this phase of many conceptual classes, the subsequent phases are used to find them for entering attributes and relations.
- It is not recommended to exclude conceptual classes from the model only because they do not appear in the requirements or do not have attributes.

Identification of conceptual classes – categorization technique

What is looking for	Example
Organisations	SalesDepartment, Airline
Incidents	Sales, Payment, Meeting, Flight, Takeoff, Landing
Processes (<i>in most cases not represented as concepts</i>)	ProductSales, SeatBooking
Rules and principles of conduct	ProductReturnRule, ReservationCancellationRule
Catalogs	ProductCatalog, PartCatalog
Results of financial activities, effects of work, contracts	Invoice, EmploymentContract, ServiceAgreement
Financial instruments and services	CreditLine, Stock
Textbooks, documents, books	Operating Instructions, DailyListPriceChange

Identification of conceptual classes – categorization technique (2)

What is looking for	Example
Physical objects	Plane, car, cash
Specifications, designs or descriptions of things	ProductSpecification, FlightDescription
Places	Airport, Shop
Transaction	Sales, Reservation
Transaction position	SalesPosition
Role of people	Clerk, pilot
Containers of other things	Outlet, basket, plane
Things in a container	Product, Passenger
Information systems and electrical/mechanical systems outside the system	PaymentAuthorisationSystem, AirTrafficControlSystem
Abstract concepts that can be described in nouns	Hunger, Agoraphobia

Identification of conceptual classes: noun phrases identification technique

- The technique is based on the identification of nouns and noun phrases in the textual (oral) descriptions of the topic and to treat them as candidates for conceptual classes and attributes.
- The full description of the use cases is an excellent description of the topic which can be used as a basis for this technique.
- Because words in the language are ambiguous and different nouns (noun phrases) may mean the same concept, it is not possible mechanical creation of noun-class assignments. This technique requires a lot of attention!
- It is recommended that the technique of identifying noun phrases be used together with the technique of using a list of categories.

Identification of conceptual classes:

Example: SalesSupport

- The use of category list and noun phrase identification techniques to analyse a use case SalesSupport can lead to the identification of the following candidates for conceptual classes :

Cashier	Customer	Manager	Sale
Product	ProductDescription	Sales	Item
Payment	CashRegister	ProductCatalogue	Shop

- There is no concept of a "correct" list!!!! It is always a set of arbitrarily chosen terms by an analyst. However, using the same techniques, it should be obtained by various analysts from similar lists.
- The problem with recognizing concepts is important and introducing them to the model on the example of a concept called Invoice:
 - Against: Repetition of information contained elsewhere in the model,
 - Pro: needed for a possible return of the product.
- Whether the concept is introduced or not, depends on iteration and whether the concept is relevant at this stage.

Domain model creation: recommended steps

- Step 1: Create a list of candidates for conceptual classes using the category list technique and the noun phrase identification technique based on the currently considered part of the requirements for the system.
- Step 2: Save the results in the diagram.
- Step 3: Add the associations needed to describe the relationships between the concepts.
- Step 4: Add the attributes necessary to record the information obtained from the requirements analysis.

Domain model: strategy for knowledge dictionary use (cartographer's strategy)

- In creating a domain model of knowledge (cartographer's strategy, or a strategy for using a dictionary of knowledge):
 - Use the vocabulary of the topic when naming classes and attributes. For example, when modelling a library, we use the terms Lender and Librarian to describe the terms Client and Library Service Employee.
 - Remove conceptual classes from the model that are considered irrelevant from the point of view of the requirements for the system at a given stage of its creation. For example, let's exclude from the model the terms Pen and Advertisement as irrelevant from the point of view of the system requirements.
 - Do not introduce to the model entities that are not present in the discussed field.

Domain model creation: mistakes

- A common mistake is representing existence as an attribute instead of a conceptual class. In order to eliminate this problem, it is recommended to use a rule:
 - If we do not think of a potential conceptual class X as a number or text in the real world, then X is probably a conceptual class, not an attribute.
- In case of doubt, in most cases we introduce an independent conceptual class.

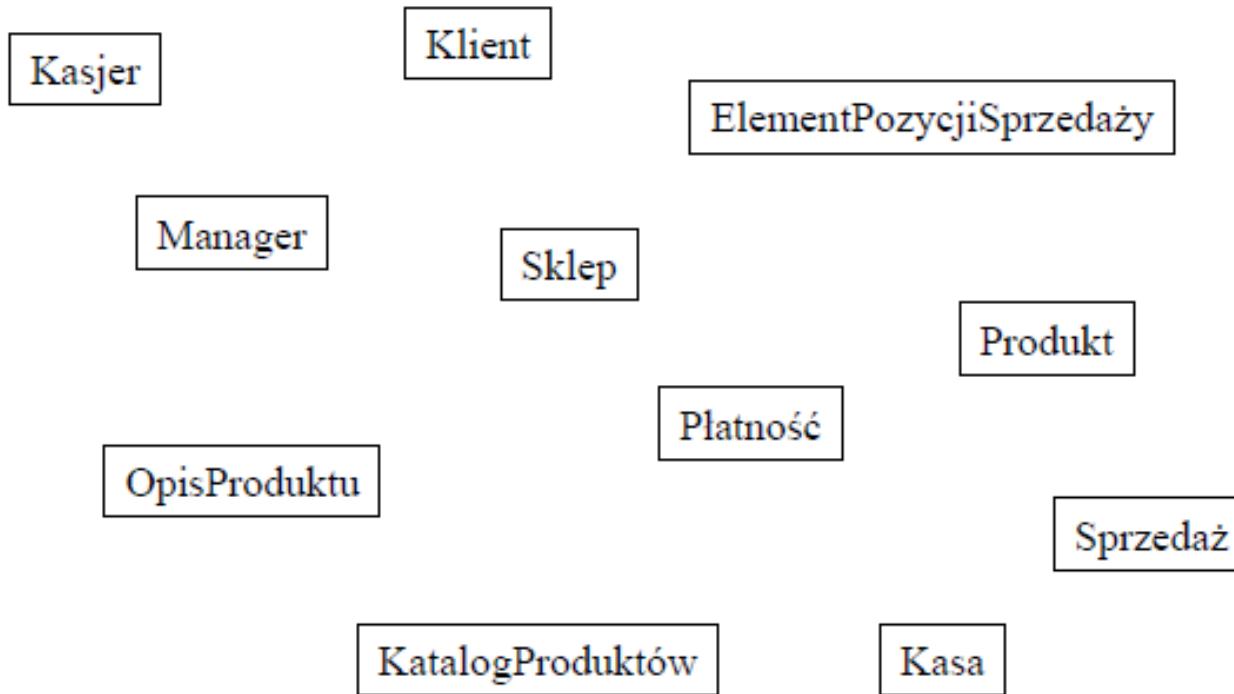
Domain model creation: „unreal world” modeling

- Many IT systems concern fields that have no direct connection with the real world, e.g. software developed in telecommunications.
- For such applications it is possible to create domain models of knowledge, but this requires a high level of abstraction and use analogies from fields for which modelling has already been done.
- For example, in telecommunications topics, the terms Message, Connection, Port, Dialogue, Road, Protocol can be used.

Reducing the representation gap

- The domain model provides us with a visual representation of a dictionary. We should draw on the model of this dictionary inspiration for naming programming entities during the phase design and implementation.
- Such an approach allows to reduce the representation gap (gap semantic) between our (i.e. analyst) model and its representation in a specific topic, and its representation in a specific topic – an IT solution.
- At the level of the domain model we use certain concepts (e.g. Sales), at the level of entity design Programming (Sales class). They are not the same entities, but the second being was created by being inspired by the first.
- This is one of the main advantages of the object-oriented approach!!!!

Example of domain model phase



Java Programming – Abstract Class & Interface

Oum Saokosal

Master's Degree in information systems, Jeonju
University, South Korea

012 252 752 / 070 252 752

oumsaokosal@gmail.com



Contact Me

- Tel: 012 252 752 / 070 252 752
- Email: oumsaokosal@gmail.com
- FB Page: <https://facebook.com/kosalgeek>
- PPT: <http://www.slideshare.net/oumsaokosal>
- YouTube: <https://www.youtube.com/user/oumsaokosal>
- Twitter: <https://twitter.com/okosal>
- Web: <http://kosalgeek.com>

Abstract Classes and Interfaces

The objectives of this chapter are:

- To explore the concept of abstract classes
- To understand interfaces
- To understand the importance of both.

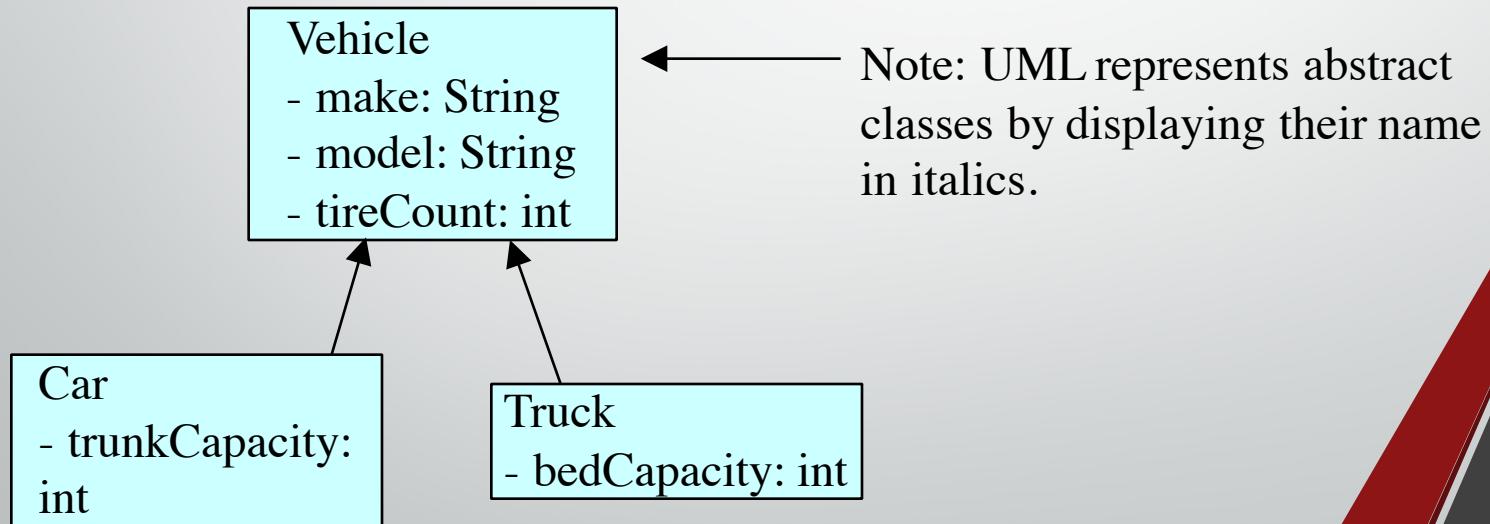
What is an Abstract class?

- Superclasses are created through the process called "generalization"
 - Common features (methods or variables) are factored out of object classifications (ie. classes).
 - Those features are formalized in a class. This becomes the superclass
 - The classes from which the common features were taken become subclasses to the newly created super class
- Often, the superclass does not have a "meaning" or does not directly relate to a "thing" in the real world
 - It is an artifact of the generalization process
- Because of this, abstract classes cannot be instantiated
 - They act as place holders for abstraction

Abstract Class Example

- In the following example, the subclasses represent objects taken from the problem domain.
- The superclass represents an abstract concept that does not exist "as is" in the real world.

Abstract superclass:



What Are Abstract Classes Used For?

- Abstract classes are used heavily in Design Patterns
 - Creational Patterns: Abstract class provides interface for creating objects. The subclasses do the actual object creation
 - Structural Patterns: How objects are structured is handled by an abstract class. What the objects do is handled by the subclasses
 - Behavioural Patterns: Behavioural interface is declared in an abstract superclass. Implementation of the interface is provided by subclasses.
- Be careful not to over use abstract classes
 - Every abstract class increases the complexity of your design
 - Every subclass increases the complexity of your design
 - Ensure that you receive acceptable return in terms of functionality given the added complexity.

Defining Abstract Classes

- Inheritance is declared using the "extends" keyword
 - If inheritance is not defined, the class extends a class called Object

```
public abstract class Vehicle
{
    private String make;
    private String model;
    private int tireCount;
    [...]
```

```
public class Car extends Vehicle
{
    private int trunkCapacity;
    [...]
```

```
public class Truck extends Vehicle
{
    private int bedCapacity;
    [...]
```

Vehicle
- make: String
- model: String
- tireCount: int

Car
- trunkCapacity: int

Truck
- bedCapacity: int

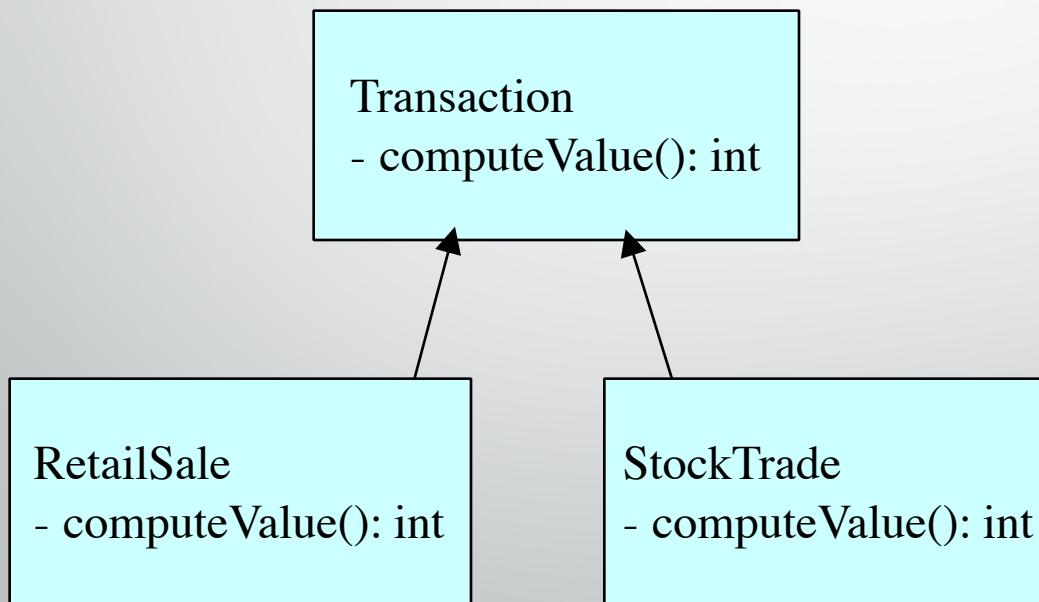
Often referred to as
"concrete" classes

Abstract Methods

- Methods can also be abstracted
 - An abstract method is one to which a signature has been provided, but no implementation for that method is given.
 - An Abstract method is a placeholder. It means that we declare that a method must exist, but there is no meaningful implementation for that methods within this class
- Any class which contains an abstract method MUST also be abstract
 - Any class which has an incomplete method definition cannot be instantiated (ie. it is abstract)
- Abstract classes can contain both concrete and abstract methods.
 - If a method can be implemented within an abstract class, and implementation should be provided.

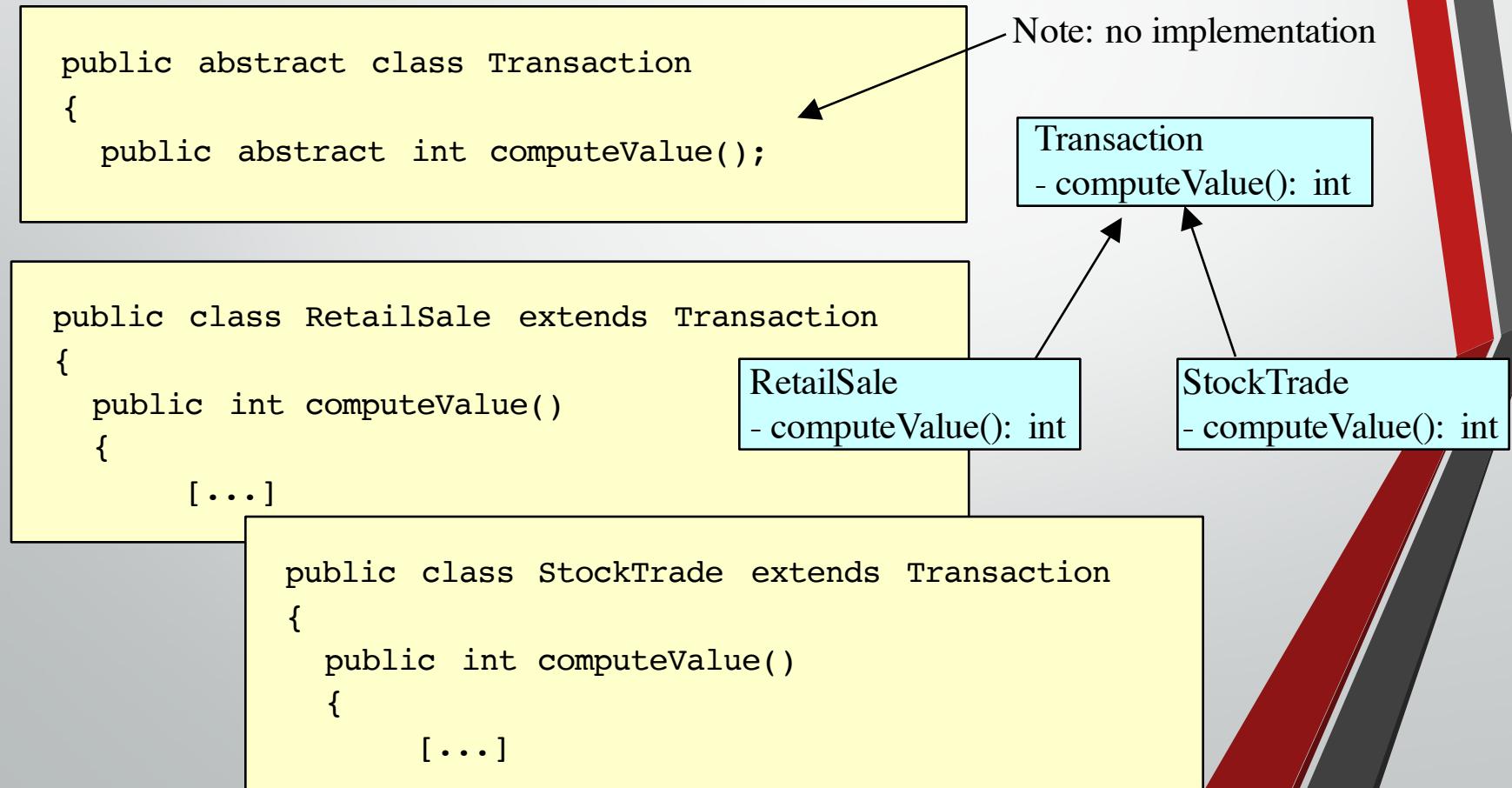
Abstract Method Example

- In the following example, a Transaction's value can be computed, but there is no meaningful implementation that can be defined within the Transaction class.
 - How a transaction is computed is dependent on the transaction's type
 - Note: This is polymorphism.



Defining Abstract Methods

- Inheritance is declared using the "extends" keyword
 - If inheritance is not defined, the class extends a class called Object



What is an Interface?

- An interface is similar to an abstract class with the following exceptions:
 - All methods defined in an interface are abstract. Interfaces can contain no implementation
 - Interfaces cannot contain instance variables. However, they can contain public static final variables (ie. constant class variables)
- Interfaces are declared using the "interface" keyword
 - If an interface is public, it must be contained in a file which has the same name.
- Interfaces are more abstract than abstract classes
- Interfaces are implemented by classes using the "implements" keyword.

Declaring an Interface

In Steerable.java:

```
public interface Steerable
{
    public void turnLeft(int degrees);
    public void turnRight(int degrees);
}
```

In Car.java:

```
public class Car extends Vehicle implements Steerable
{
    public int turnLeft(int degrees)
    {
        [...]
    }

    public int turnRight(int degrees)
    {
        [...]
    }
}
```

When a class "implements" an interface, the compiler ensures that it provides an implementation for all methods defined within the interface.

Implementing Interfaces

- A Class can only inherit from one superclass. However, a class may implement several Interfaces
 - The interfaces that a class implements are separated by commas
- Any class which implements an interface must provide an implementation for all methods defined within the interface.
 - NOTE: if an abstract class implements an interface, it NEED NOT implement all methods defined in the interface. HOWEVER, each concrete subclass MUST implement the methods defined in the interface.
- Interfaces can inherit method signatures from other interfaces.

Declaring an Interface

In Car.java:

```
public class Car extends Vehicle implements Steerable, Driveable
{
    public int turnLeft(int degrees)
    {
        [...]
    }

    public int turnRight(int degrees)
    {
        [...]
    }

    // implement methods defined within the Driveable interface
```

Inheriting Interfaces

- If a superclass implements an interface, its subclasses also implement the interface

```
public abstract class Vehicle implements Steerable
{
    private String make;
    [...]
```

```
public class Car extends Vehicle
{
    private int trunkCapacity;
    [...]
```

```
public class Truck extends Vehicle
{
    private int bedCapacity;
    [...]
```

Vehicle
- make: String
- model: String
- tireCount: int

Car
- trunkCapacity: int

Truck
- bedCapacity: int

Multiple Inheritance?

- Some people (and textbooks) have said that allowing classes to implement multiple interfaces is the same thing as multiple inheritance
- This is NOT true. When you implement an interface:
 - The implementing class does not inherit instance variables
 - The implementing class does not inherit methods (none are defined)
 - The Implementing class does not inherit associations
- Implementation of interfaces is not inheritance. An interface defines a list of methods which must be implemented.

Interfaces as Types

- When a class is defined, the compiler views the class as a new type.
- The same thing is true of interfaces. The compiler regards an interface as a type.
 - It can be used to declare variables or method parameters

```
int i;  
Car myFleet[];  
Steerable anotherFleet[];  
  
[ . . . ]  
  
myFleet[i].start();  
  
anotherFleet[i].turnLeft(100);  
anotherFleet[i+1].turnRight(45);
```

Abstract Classes Versus Interfaces

- When should one use an Abstract class instead of an interface?
 - If the subclass-superclass relationship is genuinely an "is a" relationship.
 - If the abstract class can provide an implementation at the appropriate level of abstraction
- When should one use an interface in place of an Abstract Class?
 - When the methods defined represent a small portion of a class
 - When the subclass needs to inherit from another class
 - When you cannot reasonably implement any of the methods

Czym są wzorce projektowe?

3

Przypisywanie odpowiedzialności obiektom – podstawowe zadanie fazy projektowej

- Projektowanie obiektowe jest kolejnym zadaniem, które jest realizowane po fazie zebrania wymagań oraz utworzeniu modelu wiedzy dziedzinowej, polegające na dodaniu metod do klas oraz zdefiniowaniu komunikatów przesyłanych pomiędzy obiektami w celu zrealizowania zebranych wymagań.
- Zadanie to jest kluczowe w procesie tworzenia systemów obiektowych, jednakże przypisanie metod do obiektów oraz określenie tego jak ma być realizowana interakcja pomiędzy obiektami jest niezwykle trudne.
- Przypisywanie odpowiedzialności obiektom może zostać opisane przy pomocy zestawu reguł tzw. GRASP patterns.

4

Odpowiedzialność obiektów, a metody (1)

- Odpowiedzialność to cel, zobowiązanie lub wymagane możliwości obiektu lub klasy. Odpowiedzialność jest określana jako zestaw usług realizowanych przez obiekt lub klasę.
- Odpowiedzialność obiektu za działanie (doing):
 - Realizowanie czegoś samodzielnie (tworzenie innych obiektów, wykonywanie obliczeń),
 - Inicjowanie akcji realizowanych przez inne obiekty,
 - Kontrolowanie i koordynowanie akcji innych obiektów.
- Odpowiedzialne obiektu za wiedzę (knowing)
 - Posiadanie wiedzy o danych prywatnych,
 - Posiadanie wiedzy o powiązanych obiektach,
 - Posiadanie wiedzy o faktach, które mogą zostać wywnioskowane lub obliczone.

5

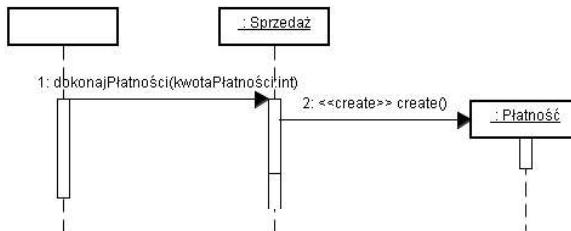
Odpowiedzialność obiektów, a metody (2)

- Przykład:
 - Odpowiedzialność obiektu za działanie: „obiekt *Sprzedaż* jest odpowiedzialny za tworzenie obiektu *ElementPozycjiSprzedaży*”,
 - Odpowiedzialność obiektu za wiedzę: „obiekt *Sprzedaż* jest odpowiedzialny za posiadanie wiedzy o sumie sprzedaży”.
- Odpowiedzialność to nie to samo co metody, ale metody są implementowane po to aby realizować odpowiedzialność obiektów.
- Odpowiedzialności obiektów są implementowane przy pomocy metod, które działają samodzielnie, bądź metod które współpracują z innymi obiektami!
- Dana odpowiedzialność może być realizowane przy pomocy bardzo różnej liczby obiektów (od kilku do kilku tysięcy)

6

Odpowiedzialność, a diagramy interakcji

- Tworzenie diagramów interakcji wspiera etap identyfikowania odpowiedzialności obiektów.
- Diagramy interakcji ilustrują wybory analityka podczas przypisywania odpowiedzialności poszczególnym obiektom.



- Przykład ilustruje przypisanie obiektemu *Sprzedaż* odpowiedzialności polegającej na tworzeniu obiektu *Płatność*. Odpowiedzialność jest realizowana podczas wywołania metody *dokonajPłatności* i jest realizowane przez tą metodę.

7

Wzorce projektowe (patterns)

- Doświadczenia projektantów systemów obiektowych zapisywane są w skodyfikowany sposób przy pomocy tzw. wzorców projektowych (patterns).
- W technologii obiektowej wzorce projektowe opisują pewien znany problem oraz jego rozwiązanie w taki sposób aby można było tą wiedzę wykorzystać w nowej sytuacji:

Nazwa wzorca projektowego

Rozwiązanie

Problem

- Wzorce projektowe opisują wiedzę sprawdzoną i powszechnie akceptowaną. Dla każdego znajdującego się reguły projektowania obiektowego wiedza przekazywana przy pomocy wzorców projektowych wydawała się będzie znajoma i podstawowa.
- Każdy wzorzec projektowy powinien posiadać sugestyną nazwę:
 - Powinna ona oddawać główne cechy wzorca, który opisuje,
 - Powinna ułatwiać komunikację podczas projektowania systemu.

8

Wzorce projektowe GRASP

- Wzorce projektowe GRASP (General Responsability Assignment Software Patterns) pozwalają na przypisywanie odpowiedzialności obiektom.
- Podstawowe wzorce projektowe typy GRASP (Larman):
 - Expert,
 - Creator,
 - High Cohesion,
 - Low Coupling,
 - Controller.

9

**Wzorce projektowe określające zasady
przypisywania odpowiedzialności obiektom**

10

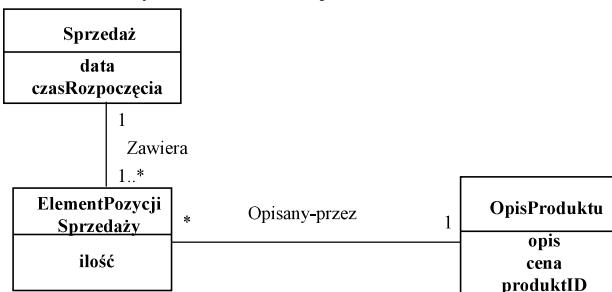
Expert: założenia

- **Rozwiążanie:** przypisuj odpowiedzialność do obiektów, które pełnią rolę ekspertów posiadających określoną informację (klasy posiadające informację niezbędną do wypełnienia pewnej odpowiedzialności).
- **Problem:** Jakie jest podstawowa zasada przypisywania odpowiedzialności obiektem?
- Podstawowy wzorzec projektowy pozwalający na przypisywanie obiektem odpowiedzialności w najbardziej oczywistych sytuacjach.
- Wzorzec zaleca, aby zacząć przypisywać obiekty swojej odpowiedzialności po wcześniejszym czytelnym ustaleniu czego dana odpowiedzialność dotyczy.
- Zasada postępowania przy poszukiwaniu klas, którym można przypisać poszukiwaną odpowiedzialność:
 - W pierwszej kolejności poszukujemy kandydatów w modelu projektowym,
 - W drugiej kolejności poszukujemy kandydatów w modelu wiedzy dziedzinowej i wprowadzamy te klasy do modelu projektowego.

11

Expert: przykład (1)

- W programie terminal wielozadaniowy POS poszukujemy klasy, która powinna znać „wartość sprzedaży razem”.
- Odpowiedzialność dotyczy określenia „kto powinien być odpowiedzialny za posiadanie informacji o wartości sprzedaży razem”.
- Przyjmujemy, że rozpoczynamy proces projektowania i nie dostępne są jeszcze, żadne klasy projektowe. Dostępny jest za to następujący model wiedzy dziedzinowej:



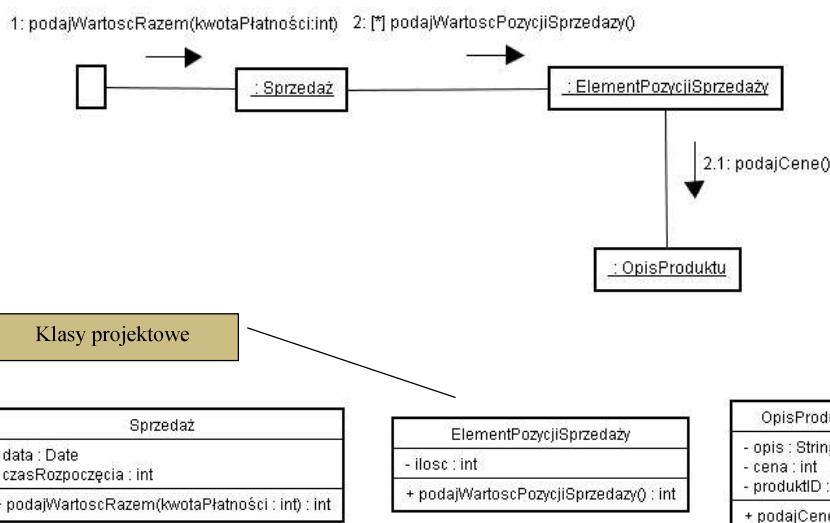
12

Expert: przykład (2)

- Przy założeniu, że rozpoczynamy proces projektowy i nie zawarte są w nim żadne klasy projektowe:
 - Kandydatem, który powinien posiadać odpowiedzialność jest *Sprzedaż*, a wprowadzoną metodą *podajWartoscRazem*,
 - W celu wyznaczenia wartości sprzedaży razem konieczna jest znajomość wartości poszczególnych sprzedanych produktów – kandydatem na przypisanie tej odpowiedzialności jest klasa *ElementPozycjiSprzedaży*, a wprowadzoną metodą *podajWartoscPozycjiSprzedaży*,
 - W celu obliczenia wartości poszczególnych sprzedanych produktów konieczna jest znajomość ceny produktów – kandydatem na przypisanie tej odpowiedzialności jest klasa *Opis produktu*, a wprowadzoną metodą *podajCene*.

13

Expert: przykład (3)



14

Expert: przykład (4)

- W procesie poszukiwania odpowiedzialności „wartość sprzedaży razem” uzyskano trzy odpowiedzialności, przypisane do trzech różnych klas:

Klasa projektowa	Odpowiedzialność
Sprzedaż	Zna wartość sprzedaży razem
ElementPozycjiSprzedaży	Zna wartość pozycji sprzedaży
OpisProduktu	Zna cenę produktu

- Odpowiedzialności zostały uzyskane w procesie budowy diagramów interakcji.
- Wzorzec projektowy Expert pozwolił na przypisanie odpowiedzialności do tego obiektu, który posiadała daną informację, niezbędną do spełnienia danej odpowiedzialności.

15

Expert: dyskusja

- Expert to elementarny i powszechnie używany wzorzec projektowy odzwierciedlający powszechną intuicję, że obiekty realizują czynności związane z informacją, którą posiadają.
- Często zdarza się, że odpowiedzialność dotyczy informacji, która jest rozproszona pomiędzy różnymi obiektami – w konsekwencji wielu ekspertów posiadających wiedzę cząstkową będzie współpracowało (przesyłając sobie komunikaty) w celu dostarczenia pełnej informacji (w przykładzie współpracują trzy klasy – trzech ekspertów posiadających informację cząstkową).
- Wzorzec ten prowadzi do projektów, w których obiekty (softwarowe) realizują pewne czynności, podczas gdy ich protoplasty w świecie rzeczywistym tego nie czynią (reguła „Do It Myself”). Np. w świecie rzeczywistym obiekt *Sprzedaż* nie jest w stanie samodzielnie obliczyć wartości razem. Ktoś inny (sprzedawca, kasa) oblicza wartość sprzedaży razem! W modelu obiektowym obiekty softwarowe są „ożywione” i dlatego realizują czynności dla informacji, którą posiadają.

16

Expert: korzyści

- Korzyści:
 - Enkapsulacja informacji jest zachowana dzięki temu, że obiekty wykorzystują swoją własną informację w celu realizowanie zadań,
 - Zachowanie jest rozproszone pomiędzy różne klasy, pozwala to tworzyć przejrzyste modele.
- Inne nazwy na wzorzec projektowy Expert:
 - „Information Expert”
 - „Do It Myself”
 - „Place responsibilities with data”
 - „That which knows, does”
 - „Put services with the Attributes They work On”

17

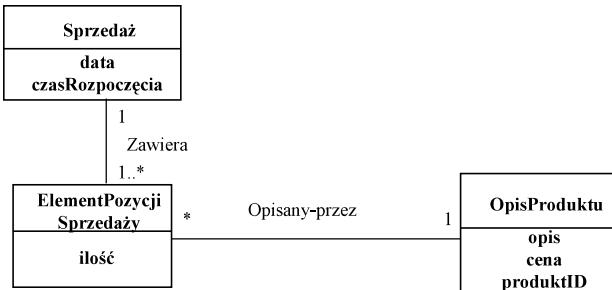
Creator: założenia

- **Rozwiążanie:** Klasa B powinna posiadać odpowiedzialność tworzenia instancji klasy A jeżeli zachodzi jedno z poniższych:
 - B agreguje obiekty klasy A,
 - B zawiera obiekty klasy A,
 - B w bardzo ścisły sposób używa obiektów A,
 - B posiada dane inicjalizujące, które zostaną przekazane do obiektu A gdy ten jest tworzony.
- **Problem:** Kto powinien być odpowiedzialny za tworzenie nowych instancji obiektów danej klasy?

18

Creator: przykład (1)

- Kto w programie wielozadaniowy POS powinien tworzyć instancje klasy *ElementPozycjiSprzedaży*?
- Analizujemy odpowiedni fragment modelu dziedzinowej:

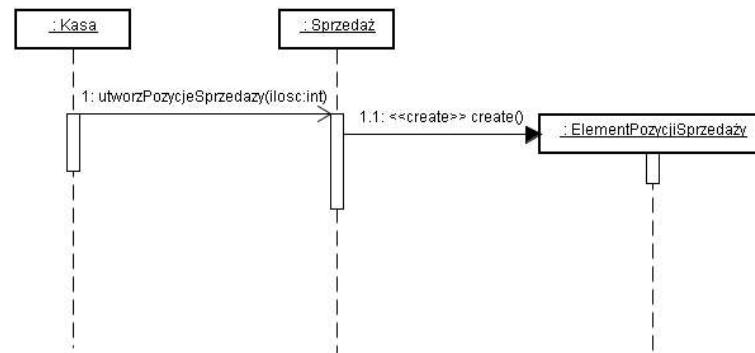


- Ponieważ *Sprzedaż* agreguje wiele obiektów *ElementPozycjiSprzedaży*, wzorzec Creator sugeruje tą właśnie klasę jako kandydata przy przypisywaniu odpowiedzialności tworzenia instancji klasy *ElementPozycjiSprzedaży*.

19

Creator: przykład (2)

- Przypisanie odpowiedzialności wymaga przypisania klasie *Sprzedaż* metody *utwórzPozycjęSprzedaży*.
- Odpowiedzialność ta została uzyskana w procesie budowy diagramu interakcji:



20

Low Coupling: założenia (1)

- **Rozwiązanie:** Odpowiedzialność powinna być przypisywana w taki sposób, aby powiązanie obiektów (coupling) było niskie.
- **Problem:** W jaki sposób utrzymać małą zależność obiektów, niski współczynnik negatywnych skutków wprowadzania zmian i zwiększać powtórne użycie?
- Powiązanie (coupling) wskazuje jak silnie jeden element jest związany z innym elementem, posiada wiedzę o innym elemencie lub zależy od innego elementu.

21

Low Coupling: założenia (2)

- Element (klasa, komponent, system) posiadający niski współczynnik powiązania (low coupling) nie zależy od zbyt wielu innych elementów. Co oznacza „zbyt wiele” zależy od rozpatrywanego kontekstu.
- Klasa z wysokim współczynnikiem powiązania zależy od zbyt wielu innych klas. Podejście to jest niekorzystne ze względu na:
 - Zmiany w powiązanych klasach wymuszają zmiany w rozpatrywanej klasie,
 - Trudności w zrozumieniu roli rozpatrywanej klasy,
 - Trudności z powtórnym użyciu – powtórne użycie zakłada każdorazowo obecność klas zależnych.

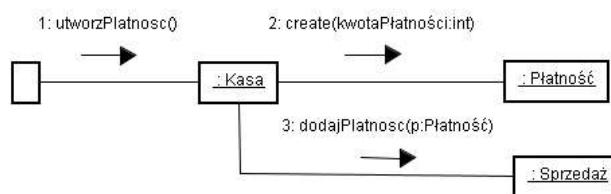
22

Low Coupling: przykład (1)

- W programie wielozadaniowy POS musimy rozstrzygnąć, kto jest odpowiedzialny za tworzenie instancji klasy *Płatność*?
- Analizujemy odpowiedni fragment z modelu wiedzy dziedzinowej:

Płatność Kasa Sprzedaż

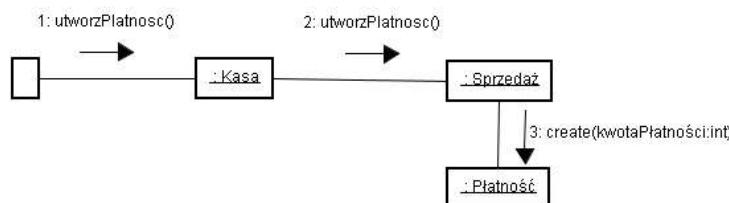
- W związku z faktem, że w świecie rzeczywistym kasa zapisuje informacje o płatnościach, wzorzec projektowy Creator sugeruje klasę *Kasa* jako kandydata do przypisania odpowiedzialności tworzenia instancji klasy *Płatność*. Następnie utworzona instancja może zostać przesłana (jako parametr) przy pomocy metody *dodajPłatność* do obiektu *Sprzedaż*.



23

Low Coupling: przykład (2)

- Przypisanie odpowiedzialności w ten sposób wiąże klasę *Kasa* z klasą *Płatność*!?
- Rozwiązaniem alternatywnym jest tworzenie klasy *Płatność* bezpośrednio przez klasę *Sprzedaż*:



- Z punktu widzenia wzorca projektowego Low Coupling powyższe rozwiązanie jest poprawne ze względu na fakt utrzymania niskiego współczynnika powiązania.
- Przykład pokazuje wykluczanie się dwóch wzorców: Creator i Low Coupling!

24

Low Coupling: powiązania, a języki programowania

- W języka C++, Java, C# spotykanymi formami powiązań w kierunku od Typu A do Typu B są:
 - A posiada atrybut, który wskazuje nainstancję B,
 - Instancja A wywołuje usługi instancji B,
 - X posiada metody, które odwołują się doinstancji B. Jest to parametr lub zmienna lokalna typu B, bądź obiekt zwracany przezdaną metodę jest typu B,
 - X jest podklassą Y,
 - Y jest interfejsem, a A implementuje ten interfejs.

25

Low Coupling: dyskusja (1)

- Wzorzec projektowy Low Coupling powinien być brany pod uwagę każdorazowo podczas podejmowania decyzji dotyczących odpowiedzialności danej klasy! Jest więc wzorcem oceniającym wyniki uzyskane po zastosowaniu innych wzorców!
- Wykorzystywanie wzorca pozwala na tworzenie klas bardziej niezależnych, redukowany jest współczynnik negatywnych skutków wprowadzania zmian.
- Wzorzec powinien być wykorzystywany razem z innymi wzorcami jako wzorzec silnie wpływający na podejmowane decyzje dotyczące przypisywania odpowiedzialności.
- Nie ma jednoznacznych zaleceń dotyczących tego, kiedy współczynnik powiązania jest zbyt wysoki. Wszystko zależy od kontekstu, decyzje są każdorazowo podejmowane arbitralnie przez projektanta.

26

Low Coupling: dyskusja (2)

- Ekstremalnym przypadkiem użycia wzorca Low Coupling jest sytuacja w której klasy nie są w ogóle powiązane. Nie jest to rozwiązanie korzystne ponieważ:
 - Łamana jest zasada, że obiekty komunikują się poprzez przesyłanie komunikatów,
 - Tworzone są pojedyncze obiekty, które są odpowiedzialne za wykonywanie całej pracy, pozostałe obiekty spełniają jedynie role zwykłych repozytoriów danych.
- Wzorzec ten jest jednym z fundamentalnych wzorców projektowych. Jego znaczenie jest podnoszone od początku istnienia analizy i projektowania obiektowego.

Wzorce projektowe określające zasady przypisywania odpowiedzialności obiektom c.d.

3

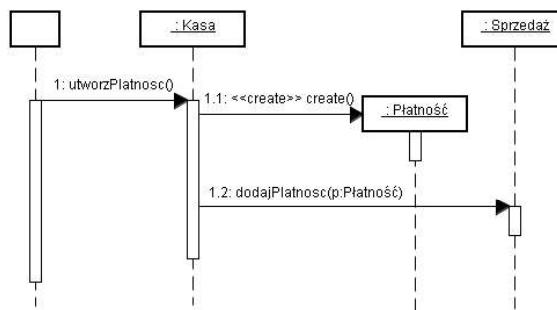
High Cohesion: założenia

- **Problem:** W jaki sposób zarządzać złożonością obiektu?
- **Rozwiązanie:** Odpowiedzialność powinna być przypisywana w taki sposób, aby spójność (cohesion) obiektu pozostawała wysoka.
- Spójność (cohesion) jest miarą na ile odpowiedzialności przypisane danemu obiekowi są ze sobą powiązane:
 - Klasa dla której odpowiedzialności są ściśle ze sobą powiązane oraz która nie ma przypisanych zbyt wielu zadań charakteryzuje się wysoką spójnością (high cohesion),
 - Klasa charakteryzująca się niską spójnością wykonuje wiele niepowiązanych ze sobą czynności, bądź wykonuje zbyt wiele zadań.
- Wady klas o niskiej spójności:
 - Trudne w powtórnym użyciu,
 - Trudne w utrzymaniu,
 - Trudno interpretowalne.
- Klasy o niskiej spójności przejmują często zadania innych klas!

4

High Cohesion: przykład (1)

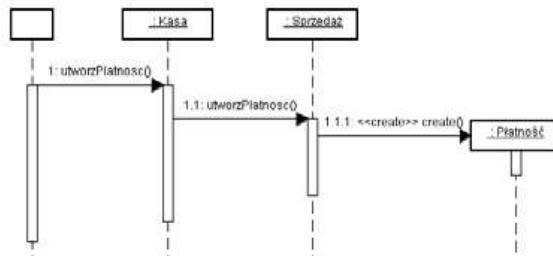
- W programie wielozadaniowy POS musimy rozstrzygnąć, kto jest odpowiedzialny za tworzenie instancji klasy *Płatność*?
- W związku z faktem, że w świecie rzeczywistym kasa zapisuje informacje o płatnościach, wzorzec projektowy Creator sugeruje klasę *Kasa* jako kandydata do przypisania odpowiedzialności tworzenia instancji klasy *Płatność*. Następnie utworzona instancja może zostać przesłana (jako parametr) przy pomocy metody *dodajPłatność* do obiektu *Sprzedaż*.



5

High Cohesion: przykład (2)

- Przyjęte rozwiązanie z punktu widzenia wzorca high cohesion jest niepoprawne w sytuacji gdy klasie *Kasa* trzeba będzie przypisać wiele odpowiedzialności. Uczyni to z tej klasy klasę o niskiej spójności.
- Wzorzec ten zaleca rozwiązanie inne, w którym płatność tworzona jest przez klasę *Sprzedaż*, co pozwala uzyskać wyższą spójność w klasie *Kasa*.



- Ponieważ drugie rozwiązanie jest zalecane przez wzorzec Low Coupling i High Cohesion to właśnie ono powinno zostać wybrane.

6

High Cohesion: dyskusja

- Podobnie jak Low Coupling wzorzec High Cohesion powinien być każdorazowo rozpatrywany przez projektanta podczas przypisywania odpowiedzialności danej klasie! Jest więc wzorcem oceniającym wyniki uzyskane przy pomocy innych wzorców.
- Klasa z wysoką spójnością posiada stosunkowo mało metod o ściśle powiązanych zadaniach. Klasa nie wykonuje zbyt wiele pracy – współpracuje z innymi obiektami w celu zlecania im wykonywania podzadań związanych z danym zadaniem.
- Wzorzec High Cohesion ma analogię do świata rzeczywistego – nie należy odciążać klasy zbyt wieloma zadaniami, tak samo jak nie należy obciążać danej osoby zbyt dużą ilością pracy. Bardziej korzystne w jednym i drugim przypadku jest delegowanie zadań.

7

High cohesion: różne poziomy spójności

- Bardzo niski poziom spójności – klasa jest samodzielnie odpowiedzialna za realizowanie zbyt wielu zadań z różnych obszarów funkcjonalnych.
- Średni poziom spójności – klasa jest samodzielnie odpowiedzialna za wykonywanie złożonych zadań z jednego obszaru funkcjonalnego.
- Wysoki poziom spójności – klasa ma przypisaną umiarkowaną ilość odpowiedzialności z jednego obszaru funkcjonalnego i współpracuje z innymi klasami w celu realizacji określonego zadania.
- Umiarkowany poziom spójności – klasa ma przypisaną umiarkowaną ilość odpowiedzialności w różnych obszarach funkcjonalnych, które te obszary są związane z klasą, ale nie są powiązane pomiędzy sobą.

8

Controller: założenia (1)

- **Problem:** Jaka klasa powinna być odpowiedzialna za obsługę zdarzeń systemowych?
- **Rozwiązanie:** Odpowiedzialność dotycząca obsługi komunikatów systemowych powinna być przypisywana jednej z następujących klas:
 - Reprezentującej cały system, urządzenie lub podsystem (kontroler fasadowy – facade controller)
 - Reprezentującej jeden przypadek użycia w ramach którego obsługiwane są pewne zdarzenia systemowe (kontroler przypadku użycia – use case controller). Kontrolery tego typu przyjmują nazwy:
<NazwaPrzypadkuUżycia>Handler,
<NazwaPrzypadkuUżycia>Coordinator,
<NazwaPrzypadkuUżycia>Session.

9

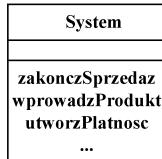
Controller: założenia (2)

- **Zdarzenie systemowe** to zdarzenie generowane przez aktorów zewnętrznych względem rozpatrywanego modelu.
- Zdarzenia takie są powiązane z **operacjami systemowymi**, czyli operacjami systemu wykonywanymi w odpowiedzi na zdarzenia systemowe.
- Kontroler to obiekt nie będący z poziomu interfejsu użytkownika, ale odpowiedzialny za obsługę zdarzeń systemowych. Kontroler definiuje metody umożliwiające realizację operacji systemowych.
- Kontroler to element składowy wzorca MVC (model-view-controller).

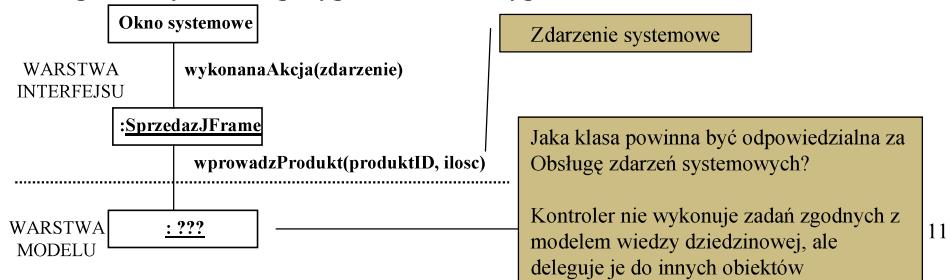
10

Controller: przykład (1)

- W aplikacji wielozadaniowej POS można wyróżnić następujące operacje systemowe przypisane roboczo do klasy *System*:



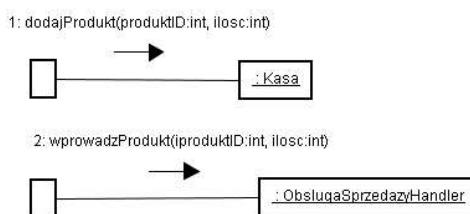
- Powyższe nie oznacza, że klasa *System* zostanie wprowadzona do modelu. Odpowiedzialności związane z operacjami systemowymi powinny zostać przypisane klasie typu kontroler.



11

Controller: przykład (2)

- Wzorzec Controller sugeruje następujących kandydatów na klasę kontrolera:
 - Klasa reprezentująca cały system: *Kasa*, *POSSystem*,
 - Klasa obsługująca zdarzenia systemowe związane z jednym przypadkiem użycia: *ObslugaSprzedazyHandler*.
- Diagramy interakcji dla tych klas są następujące:

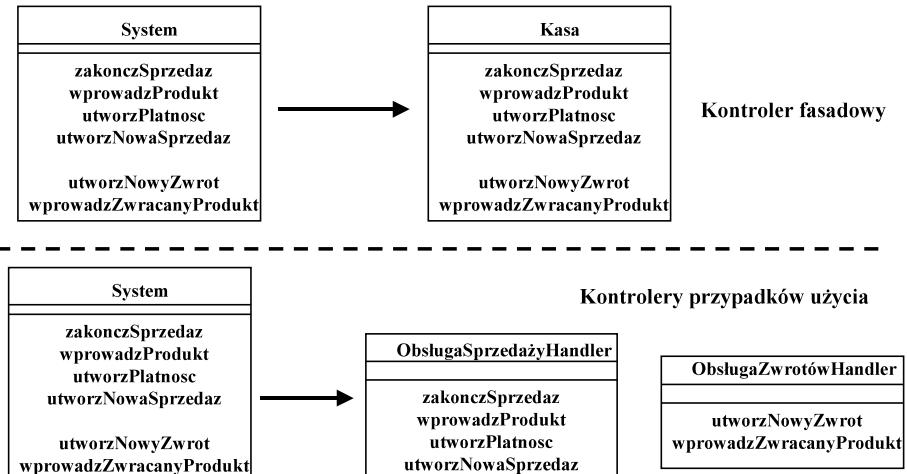


- To która z tych dwóch klas zostanie wybrana jako najbardziej odpowiedni kontroler zależy od rozpatrywanego kontekstu.

12

Controller: przykład (3)

- Podczas fazy projektowej operacje systemowe zostają przypisane do jednego, bądź więcej kontrolerów:



13

Controller: dyskusja (1)

- Wzorzec Controller dostarcza rozwiązań na obsługę komunikatów różnego rodzaju, w szczególności tych generowanych przez GUI.
- Kontroler powinien delegować zadania do innych obiektów, jego rola powinna polegać na koordynacji i kontroli. Sam nie powinien realizować zbyt wiele.
- Kontrolery fasadowe powinny zostać wprowadzane jeżeli nie ma zbyt wiele komunikatów do obsłużenia lub interfejs użytkownika nie może przekierowywać obsługi komunikatów do wiele kontrolerów.
- Kontrolery przypadków użycia powinny być wprowadzane jeżeli jest wiele komunikatów rozdzielonych pomiędzy różne procesy systemowe. Kontrolery takie mają charakter czysto techniczny, nie znajdujemy dla nich pokrycia w modelu wiedzy dziedzinowej.

14

Controller: dyskusja (2)

- Wzorzec Controller wyraża koncepcję, która głosi że klasy interfejsu użytkownika (okna, przyciski, itp.) nie powinny posiadać odpowiedzialności związanych z obsługą zdarzeń systemowych. Operacje systemowe powinny być obsługiwane na poziomie modelu dziedzinowego, a nie na poziomie interfejsu!!!
- Kontroler otrzymuje komunikaty z poziomu interfejsu użytkownika i koordynuje wypełnienie zadań w nich wyrażonych poprzez delegowanie zadań do innych obiektów.
- Korzyści:
 - Zwiększenie możliwości powtórnego użycia kodu - łatwość przenoszenia kodu pomiędzy różne narzędzia (biblioteki) do tworzenia interfejsu użytkownika,
 - Możliwość prostej weryfikacji realizacji danego przypadku użycia np. weryfikacja poprawnej kolejności obsługi komunikatów.

15

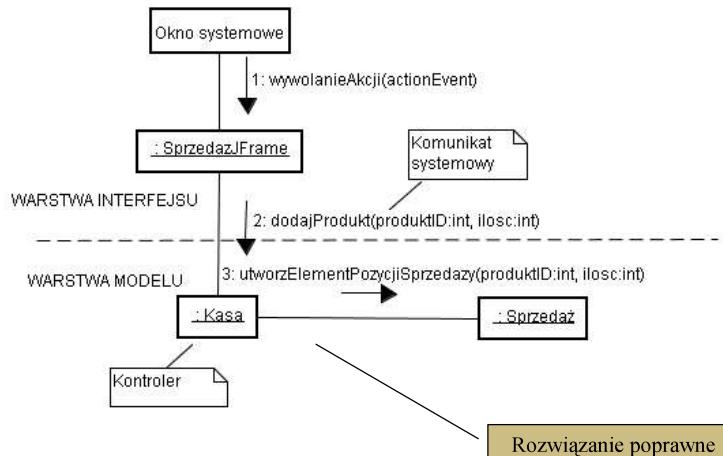
Controller: przeładowane kontrolery

- Przeładowane kontrolery (bloated controller) to źle zaprojektowane kontrolery posiadające niską spójność tzn. posiadające zbyt dużo odpowiedzialności. Sytuacja taka zachodzi gdy:
 - Wprowadzony jest jedynie jeden kontroler otrzymujący wszystkie komunikaty systemowe, których jest wiele,
 - Kontroler wykonuje samodzielnie dużo zadań, aby zrealizować komunikaty systemowe, zamiast delegować zadania do innych obiektów,
 - Kontroler posiada wiele atrybutów i przechowuje istotne dla danego problemu informacje, które powinny być przechowywane w innych obiektach, bądź duplikuje informacje przechowywane w innych obiektach.
- Rozwiązanie problemu:
 - Wprowadzenie licznych kontrolerów,
 - Projektowanie kontrolerów, które poprawnie delegują zadania do innych obiektów

16

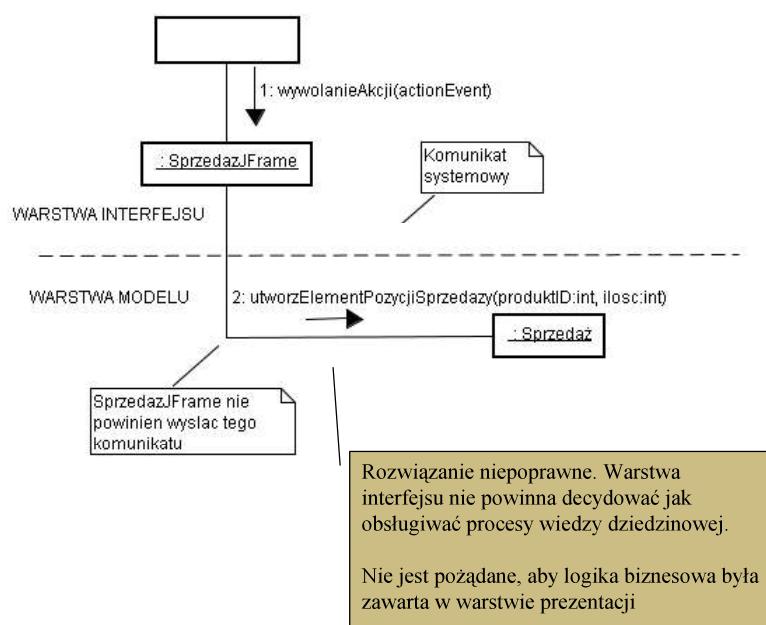
Controller: interfejs użytkownika i kontrolery (1)

- Klasy interfejsu użytkownika nie powinny posiadać odpowiedzialności związanych z obsługą zdarzeń systemowych:



17

Controller: interfejs użytkownika i kontrolery (2)



18

Widoczność

19

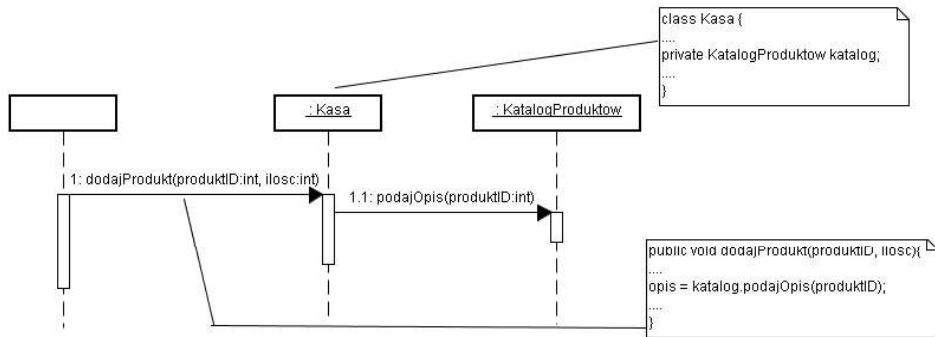
Wprowadzenie

- Widoczność to zdolność jednego obiektu do widzenia, bądź posiadania referencji do drugiego obiektu.
- Aby wysłać komunikat od obiektu nadawcy do obiektu odbiorcy konieczne jest, aby odbiorca był widoczny dla nadawcy tzn. aby nadawca miał referencję, bądź wskaźnik do obiektu odbiorcy.
- Występują następujące rodzaje widoczności:
 - Widoczność atrybutowa (attribute visibility) – B jest atrybutem A,
 - Widoczność parametryczna (parameter visibility) – B jest parametrem metody A,
 - Widoczność lokalna (local visibility) – B jest lokalnym (nie jest parametrem) obiektem w metodzie A,
 - Widoczność globalna (global visibility) – B jest widoczne globalnie.
- Najbardziej typowym rodzajem widoczności jest sytuacja, w której referencja, bądź wskaźnik do jednego obiektu jest atrybutem w innym obiekcie.

20

Widoczność atrybutowa

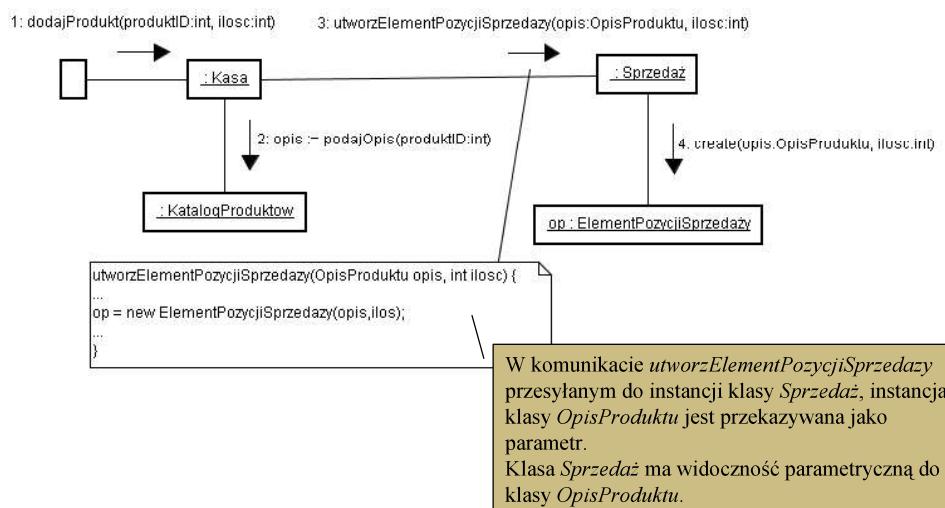
- Widoczność atrybutowa zachodzi gdy obiekt B jest atrybutem obiektu A. Widoczność jest trwała ponieważ istnieje tak długo jak istnieją obiekty A i B .
- Jest to najczęściej spotykany rodzaj widoczności.



21

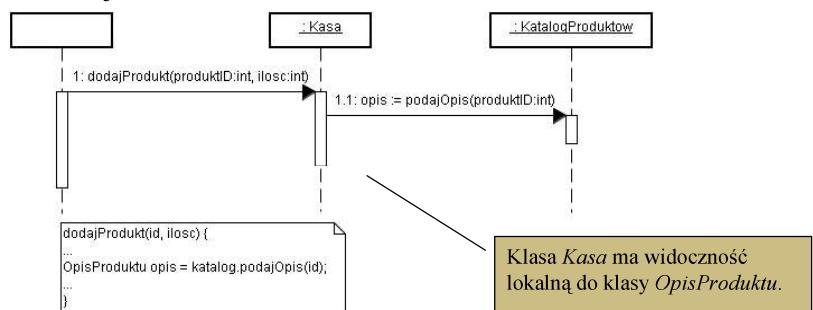
Widoczność parametryczna

- Widoczność parametryczna zachodzi wtedy, gdy obiekt B jest przekazywany jako parametr do metody A. Widoczność ma charakter czasowy ponieważ trwa tak długo jak wykonywana jest metoda A.



Widoczność lokalna

- Widoczność lokalna z klasy A do klasy B zachodzi wtedy, gdy B jest deklarowane jako lokalny obiekt w metodzie klasy A. Widoczność ma charakter czasowy ponieważ trwa tak długo jak wykonywana jest metoda A.
- Widoczność ta może zostać uzyskana poprzez:
 - Utworzenie nowego obiektu lokalnego i przypisania go do zmiennej lokalnej,
 - Przypisania zmiennej lokalnej obiektu zwracanego przez wywoływaną metodę.



23

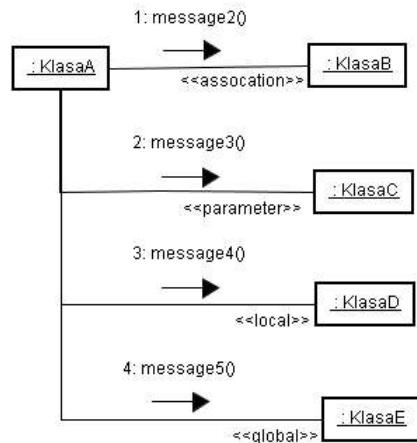
Widoczność globalna

- Widoczność globalna pomiędzy klasą A, a klasą B istnieje wtedy gdy B jest globalnie dostępne dla klasy A. Widoczność jest trwała ponieważ istnieje tak długo jak istnieją obiekty A i B.
- Podejście to jest najrzadziej spotykane w programowaniu obiektowym!
- Jednym sposobem uzyskania tego typu widoczności jest przypisanie instancji klasy B do zmiennej globalnej (dostępne w języku C++).

24

Widoczność i UML

- UML pozwala na wyrażenie rodzaju widoczności w diagramach kolaboracji. Notacja ta jest używana opcjonalnie w sytuacjach, w których informacja o rodzaju widoczności jest istotna.



Wzorce projektowe „Bandy czworga”

3

Czym są wzorce projektowe „Bandy Czworga”?

- Wzorce projektowe „Bandy Czworga” to wzorce projektowe opisane przez czterech autorów (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides) w „Design Patterns Elements of Reusable Object-Oriented Software” w 1995 roku.
- Opracowanie zawiera 23 popularne wzorce projektowe oceniane jako niezwykle przydatne podczas projektowania obiektowego.
- Przyjmuje się, że w praktyce z 23 przedstawionych wzorców ok. 15 jest szeroko stosowanych.
- Wzorce projektowe podzielone są na następujące kategorie:
 - Wzorce kreacyjne (creational patterns): Factory, Builder, Factory Method, Prototype, Singleton,
 - Wzorce strukturalne (structural patterns): Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy,
 - Wzorce behawioralne (behavioral patterns): Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

4

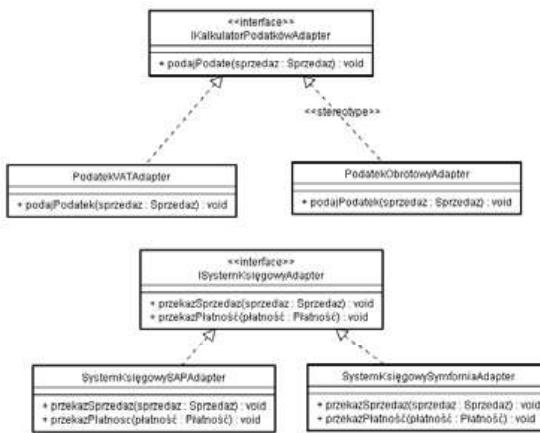
Adapter: założenia

- **Problem:** W jaki sposób należy rozwiązać problem niekompatybilnych interfejsów (sprzęgów)? W jaki sposób należy udostępniać stabilny interfejs (sprzęg) do podobnych komponentów o różnych interfejsach.
- **Rozwiązanie:** Zalecane jest wprowadzenie nowego obiektu pośredniego (adapter), który konwertuje niekompatybilny interfejs w interfejs, który jest oczekiwany przez klienta.
- Wzorzec umożliwia współpracę dwóch klas w sytuacji, w której klient nie ma możliwości użycia wprost interfejsu danej klasy. Ma to miejsce w sytuacji w której klasa klient współpracuje z wieloma aplikacjami zewnętrznymi do których dostęp jest możliwy przy pomocy różnych (i niekompatybilnych) interfejsów (różne API).

5

Adapter: przykład

- W programie wielozadaniowy POS zachodzi potrzeba współpracy z wieloma systemami (aplikacjami) zewnętrznymi: kalkulatory podatków, systemy autoryzacji kart kredytowych, system księgowy, itp. Każdy z tych systemów udostępnia różny interfejs w różnych technologiach (RMI, SOAP).



6

Adapter: dyskusja

- Adaptry są budowane przy wykorzystaniu składniowego mechanizmu interfejsów (Java), bądź składniowego elementu polimorfizmu (C++, Java).
- Korzystanie z wzorca Adapter jest zalecane w następujących sytuacjach:
 - Niezbędne jest wykorzystanie istniejącej klasy, której interfejs nie może być wykorzystany przez klasę klienta,
 - Chcemy utworzyć klasę wykorzystywaną wielokrotnie (reusable), która współpracuje z niepowiązanymi oraz niedospecyfikowanymi klasami tzn. z klasami, które nie mają identycznych interfejsów.

7

Factory: założenia i przykład

- **Problem:** Kto powinien być odpowiedzialny za tworzenie obiektów w sytuacji w której należy uwzględnić dodatkowe ograniczenia np. duża złożoność tworzonych struktur.
- **Rozwiązanie:** Wprowadzenie obiektu Factory, który jest odpowiedzialny za tworzenie rozpatrywanych obiektów.
- Zalety z wprowadzonego rozwiązania:
 - Przekazanie odpowiedzialności tworzenia skomplikowanych struktur do specjalizowanych i spójnych obiektów,
 - Ukrywanie złożoności procesu tworzenia skomplikowanych struktur,
 - Możliwość wprowadzenia technik optymalizacji zarządzania pamięcią (cachowanie).

8

Factory: przykład

- **Przykład:** W programie wielozadaniowy POS konieczne jest utworzenie adapterów *PodatekVATAdapter*, *SystemKsięgowySAPAdapter*. Kto powinien być odpowiedzialny za tworzenie tych obiektów?
- Przypisanie odpowiedzialności tworzenia obiektów do klas z modelu wiedzy dziedzinowej doprowadzi do przekroczenia zakresu odpowiedzialności tych klas (wykonywanie zadań związanych z rozpatrywaną dziedziną np. obliczanie podatków).
- Zaleca się rozdzielenie odpowiedzialności zgodnie z grupami tematycznymi i utworzenie obiektu *UsługiFactory* typu Factory .



9

Singleton: założenia

- **Problem:** Konieczne jest, aby istniała dokładnie jedna instancja danej klasy („singleton”) oraz aby była ona globalnie dostępna.
Rozwiązanie: Wprowadzenie metod statycznych do klasy, która zwraca instancje klasy „singleton”.
- Wzorzec Singleton zakłada, że klasa jest sama odpowiedzialna za dostarczanie swojej jedynej instancji! Klasa jest tak skonstruowana, że gwarantuje, że inne jej instancje nie będą utworzone.
- Wzorzec Singleton jest zalecany, gdy w programie ma być dostępna jedynie jedna instancja danej klasy i musi być ona dostępna dla klientów ze znanego wszystkim miejsca.
- Przykładem klasy tego typu może być np. printer spooler. W większości programów istnieje tylko jedna instancja tej klasy, która powinna być dostępna z całego programu.

10

Singleton: realizacja

- Klasa wprowadza metodę *getInstance*, która jako jedyna może być wykorzystana do uzyskania instancji „singleton”. W większości języków programowania metoda ta będzie metodą statyczną (C++, Java). Jako statyczny wprowadzony zostaje również atrybut przechowującyinstancję.
- Definicja klasy wygląda następująco:

```
public class Singleton {  
    private static Singleton instance;  
    public static Singleton getInstance() {  
        if (instance == null)  
            instance = new Singleton();  
        return instance;  
    }  
}
```

11

Singleton: przykład (1)

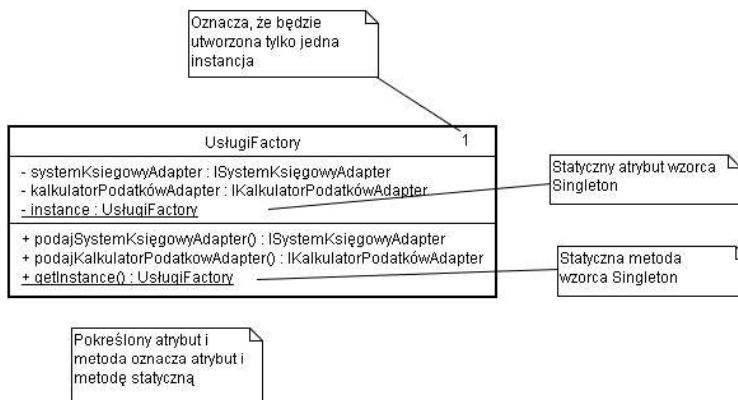
```
public class Singleton {  
    private static Singleton instance;  
    private int Liczba = 0;  
  
    public static Singleton getInstance() {  
        if (instance == null)  
            instance = new Singleton();  
        return instance;  
    }  
  
    public int obliczeniaDowolne(){  
        Liczba = Liczba +1;  
        return Liczba;  
    }  
}  
  
class Start  
{  
    public static void main(String args[]){  
        int i;  
        i = Singleton.getInstance().obliczeniaDowolne();  
        System.out.println(i);  
        i = Singleton.getInstance().obliczeniaDowolne();  
        System.out.println(i);  
    }  
}
```

Odwołanie się do
jedynej instancji klasy

12

Singleton: przykład (2)

- Wzorzec Singleton powinien być wykorzystywany do tworzenia obiektów typu Factory – w każdym programie powinien być jeden taki obiekt.
- W programie wielozadaniowym POS klasa *UslugiFactory* powinna być klasą utworzoną zgodnie z wzorcem Singleton:



13

Singleton: dyskusja (1)

- Wprowadzenie wzorca Singleton jest lepszym rozwiązaniem niż wykorzystanie samej zmiennej globalnej, bądź obiektu statycznego:
 - Zmienne globalne są co prawda globalnie dostępne, ale nie można zapobiec wielokrotnemu ich tworzeniu,
 - Podczas tworzenia programu (faza inicjalizacji zmiennych globalnych) możemy nie posiadać pełnej informacji potrzebnej podczas tworzenia takiej instancji (singleton może wymagać danych, które zostaną obliczone w trakcie działania programu).
- Rozwiązania przyjęte we wzorcu (tzn. wprowadzenie metody statycznej zwracającej instancję klasy singleton) jest bardziej korzystne niż wprowadzenie wszystkich metod jako statyczne (np. uczynienie jako statyczną metodę *podajSystemKsięgowyAdapter()*):
 - Możliwe jest tworzenie podklas na bazie klasy singleton (gdyby wszystkie metody były statyczne nie jest to możliwe – metody statyczne w większości języków nie są polimorficzne),
 - Klasa nie zawsze jest klasą singleton we wszystkich kontekstach użycia;¹⁴ przyjęte rozwiązanie umożliwia proste wprowadzanie zmian.

Singleton: dyskusja (2)

- We wzorcu został wykorzystany mechanizm tzw. późnej inicjalizacji (lazy initialization) w przeciwieństwie do możliwej tzw. pochopnej inicjalizacji (eager initialization):

Późna inicjalizacja:

```
public class Singleton {  
    private static Singleton instance;  
    public static Singleton getInstance() {  
        if (instance == null)  
            instance:=new Singleton;  
        return instance;  
    }  
}
```

Pochopna inicjalizacja:

```
public class Singleton {  
    private static Singleton instance =new Singleton;  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

- Późna inicjalizacja jest preferowana ze względu na:
 - Instancja nie jest tworzona w sytuacji, w której nie jest wykorzystywana; unikamy zbędnej pracy tworzenia niepotrzebnych struktur,
 - Inicjalizacja w `getInstance` może zostać wykorzystana do zrealizowanie złożonych i warunkowych operacji.

15

Strategy: założenia

- **Problem:** W jaki sposób modelować klasy, które są powiązane, ale różnią się co do realizowanych algorytmów? W jaki sposób modelować, aby istniała możliwość elastycznej zmiany tych algorytmów?
- **Rozwiązanie:** Zdefiniować każdy z algorytmów w oddzielnych klasach, ale posiadających wspólny interfejs.
- Wzorzec Strategy znajduje zastosowanie gdy nie jest korzystne umieszczanie wszystkich algorytmów w jednej klasie:
 - Jeżeli umieścilibyśmy dany algorytm w klasie klienckiej (tej która z nich korzysta), klasa ta stałaby się większa i trudna w utrzymaniu, w szczególności wtedy gdy wykorzystywałaby wiele algorytmów,
 - Różne algorytmy są wykorzystywane w różnym czasie - nie jest korzystne umieszczanie wszystkich algorytmów w jednej klasie w sytuacji w której z nich nie korzystamy,

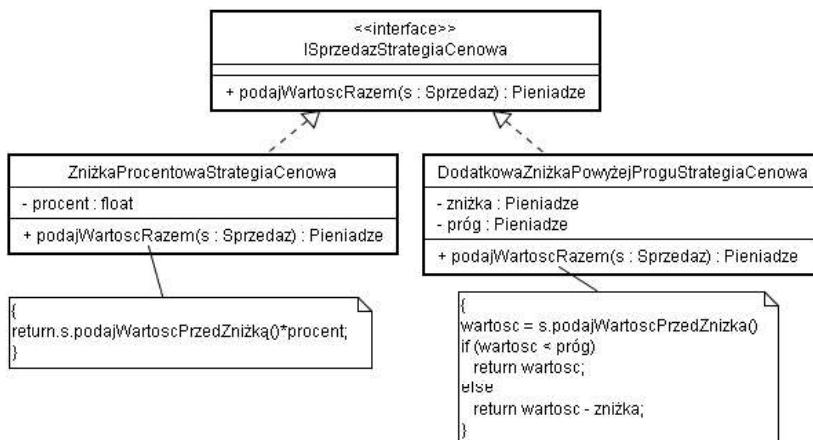
16

Strategy: przykład (1)

- W programie wielozadaniowy POS konieczne jest uwzględnienie obsługi złożonej polityki cenowej sklepu (zniżki dla różnych grup klientów). Polityka ta zmienia się w czasie (w jednym okresie zniżki mogą wynosić 5%, w innym, 15%).
- Rozwiązaniem problemu jest zastosowanie wzorca Strategy poprzez utworzenie licznych klas implementujących poszczególne algorytmy posiadających polimorficzną metodę *podajWartoscRazem*.
- Każda z metod *podajWartoscRazem* posiada jako parametr klasę *Sprzedaz*, aby przed obliczeniem zniżki mogła pobrać sumę sprzedaży przed zniżką (metoda *podajWartoscPrzedZnizka* z klasy *Sprzedaz*). Implementacja każdej z metod *podajWartoscRazem* będzie różna – uwzględniane będą różne algorytmy obliczania zniżek.
- Przyjęte rozwiązanie pozwala uniknąć sytuacji, w której metody z różnymi rodzajami zniżek są umieszczane wprost w klasie *Sprzedaz*!

17

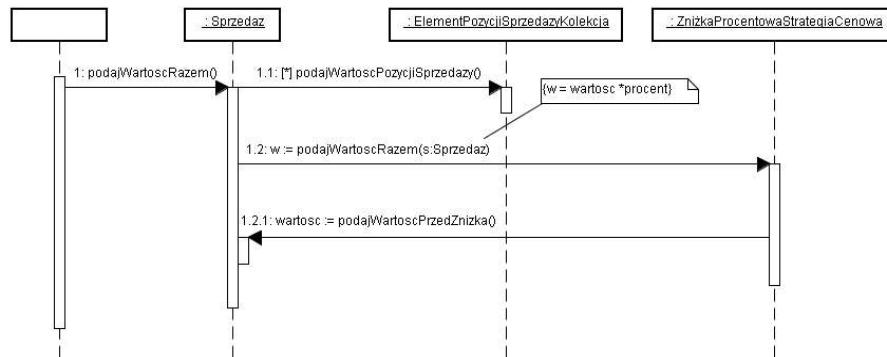
Strategy: przykład (2)



18

Strategy: dyskusja (1)

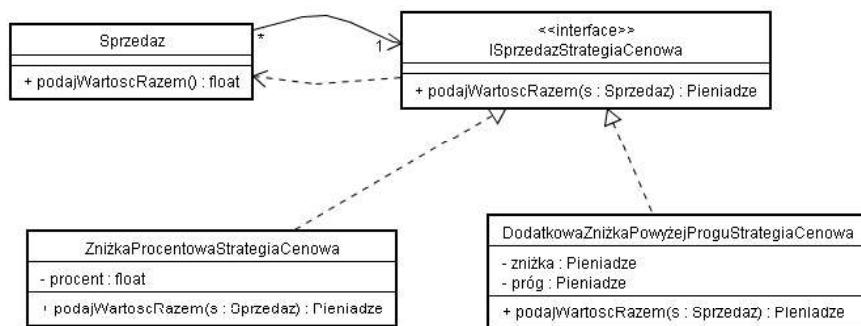
- Obiekt realizujący wzorzec Strategy jest zawsze powiązany z tzw. obiektem kontekstowym (context object), czyli obiektem klienckim.
- W rozpatrywanym przykładzie obiektem tym jest instancja klasy *Sprzedaż*. Obiekt ten deleguje część pracy na obiekt Strategy w sytuacji w której otrzymuje zadanie obliczenia sumy zakupów (metoda *podajWartoscRazem*).



19

Strategy: dyskusja (2)

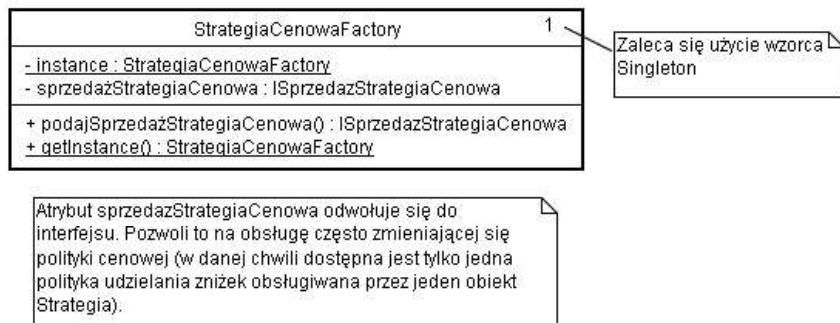
- Rozwiązanie wykorzystane w przykładzie tzn. przekazanie instancji klasy *Sprzedaż* do obiektu Strategy jest rozwiązaniem często spotykanym. Oznacza to, że obiekt Strategy posiada widzialność parametryczną do obiektu kontekstowego. Ponadto obiekt kontekstowy posiada widoczność atrybutową do obiektu Strategy.



20

Strategy: dyskusja (3)

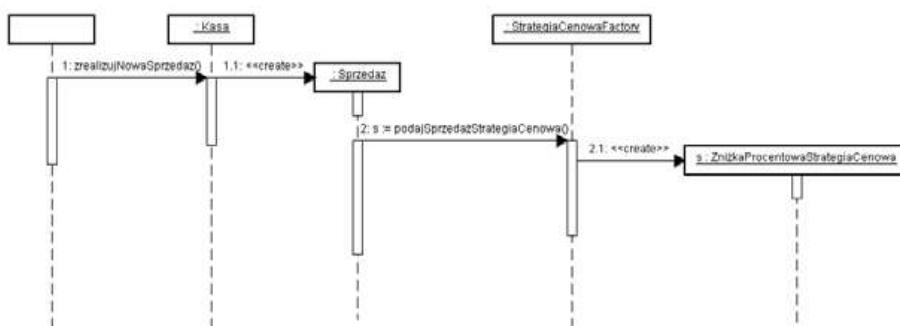
- Wzorzec Factory zaleca, aby za tworzenie obiektów Strategy był odpowiedzialny obiekt Factory. Ze względu na wzorzec High Cohesion zaleca się, aby były tworzone różne obiekty Factory realizujące różne zadania.
- W związku z powyższym w rozpatrywanym przykładzie zaleca się utworzenie nowej klasy *StrategiaCenowaFactory* służącej do tworzenia wszystkich obiektów typu Strategy.



21

Strategia: dyskusja (4)

- Podczas tworzenia instancji *Sprzedaz* kierowane będzie pytanie do instancji *StrategiaCenowaFactory*, która zwróci obiekt obsługujący aktualnie obowiązującą strategię cenową.



22

Wzorce projektowe „Bandy czworga” (c.d.)

3

Composite: założenia

- **Problem:** W jaki sposób przetwarzać grupy obiektów (obiekty grupujące inne obiekty) w taki sam sposób jak obiekty występujące samodzielnie?
- **Rozwiązanie:** Zdefiniować klasy posiadające ten sam interfejs dla grup obiektów i obiektów występujących samodzielnie.
- Wzorzec Composite znajduje zastosowanie gdy:
 - Chcemy reprezentować hierarchie obiektów typu całość-część,
 - Chcemy aby klient nie postrzegał różnic pomiędzy obiektami grupującymi różne obiekty, a obiektami występującymi samodzielnie.
- Rozwiązanie jest korzystne w sytuacji, w której konieczne jest obsłuszenie różnych strategii (algorytmów), które mogą pozostawać w sprzeczności ze sobą.

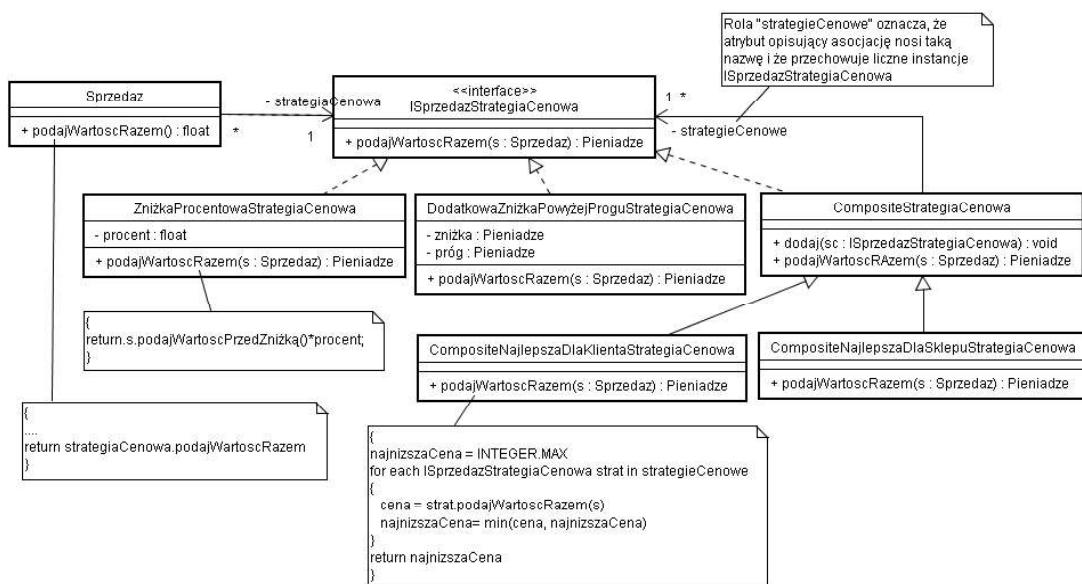
4

Composite: przykład (1)

- W programie wielozadaniowy POS konieczne jest obsłużenie sytuacji w której funkcjonuje jednocześnie wiele strategii cenowych mogących wykluczać się między sobą. Np. rozpatrujemy sytuację, w której na cenę mogą wpływać jednocześnie trzy czynniki:
 - Dzień tygodnia (początek tygodnia tańczy, niż weekend),
 - Rodzaj klienta (klient stał ma zniżki),
 - Rodzaj produktu (system zniżek dla danego produktu).
- W jaki sposób mamy zaprojektować system, aby obiekt *Sprzedaż* nie musiał wiedzieć o tym, czy w danej chwili obowiązuje jedna czy wiele strategii cenowych?
- Rozwiązaniem jest utworzenie klasy *CompositeNajlepszaDlaKlientaStrategiaCenowa*, która implementuje interfejs *ISprzedazStrategiaCenowa* i sama zawiera również instancje klasy *ISprzedazStrategiaCenowa*!

5

Composite: przykład (2)



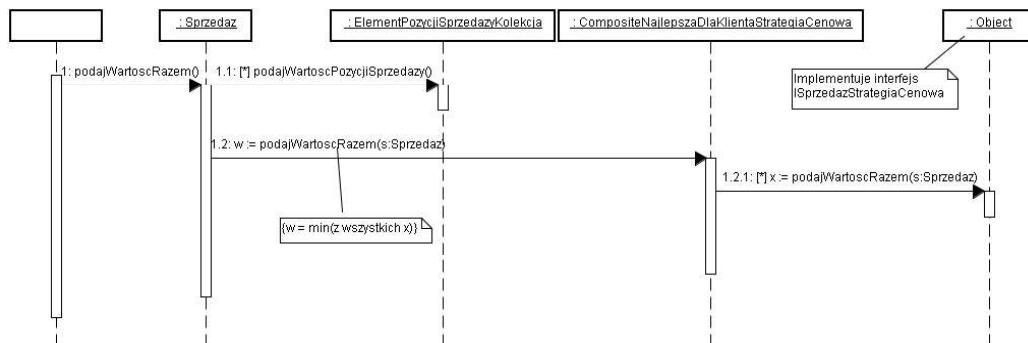
6

Composite: przykład (3)

- Charakterystyczne dla wzorca jest to, że obiekt grupujący (zewnętrzny), jak i grupowany (wewnętrzny) implementują ten sam interfejs.
- Dzięki przyjętemu rozwiążaniu (*CompositeNajlepszaDlaKlientaStrategiaCenowa* posiada interfejs *ISprzedazStrategiaCenowa*) instancja klasy *Sprzedaz* może przetwarzać obiekty złożone (*CompositeNajlepszaDlaKlientaStrategiaCenowa*) grupujące inne obiekty, bądź obiekty indywidualne (*ZniżkaProcentowaStrategiaCenowa*). Dla instancji *Sprzedaz* nie jest istotne, czy strategia cenowa jest złożona, czy atomowa.

7

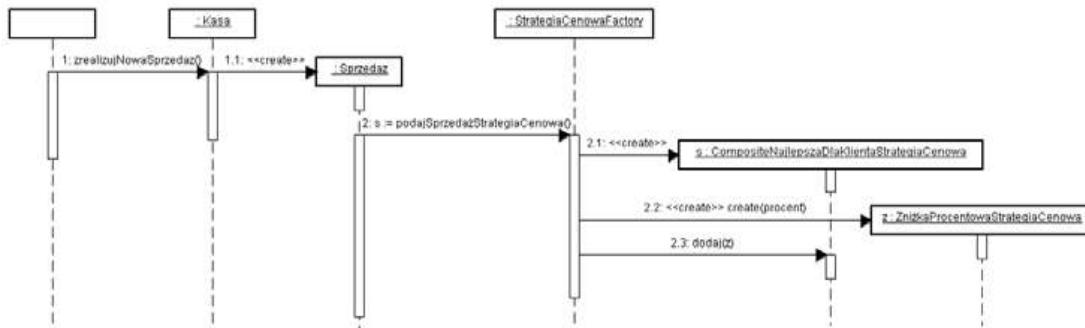
Composite: przykład (4)



8

Composite: przykład (5)

- Obiekty wewnętrzne dodawane są do obiektu typu *Composite* w momencie w którym realizowana jest sprzedaż (tworzona jest instancja klasy *Sprzedaż*)



9

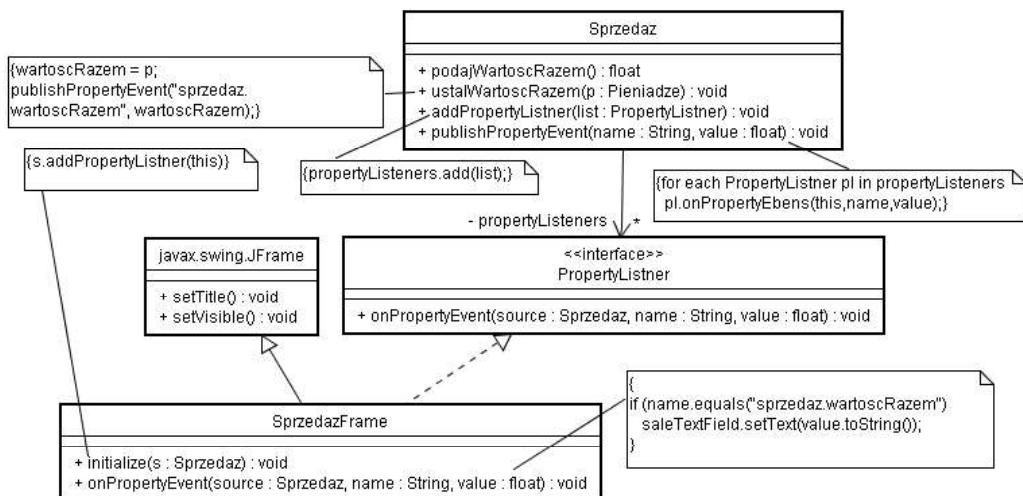
Observer: założenia

- **Problem:** W jaki sposób modelować sytuację, w której obiekty (słuchacze) zainteresowane śledzeniem zmian zachodzących w stanie pewnego obiektu (publikator) chcą reagować w różny sposób na zmianę generowaną przez ten obiekt.
- **Rozwiązanie:** Zalecane jest zdefiniowanie interfejsu „listener” (lub inaczej „subscriber”), który powinien być implementowany przez obiekty słuchacze. Publikator powinien mieć możliwość rejestracji słuchaczy i informowania ich o zachodzących zmianach.
- Wzorzec znajduje zastosowanie do odświeżania zawartości GUI w odpowiedzi na zmianę określonych danych.
- Dzięki wzorcowi nie jest łamana zasada rozdzielenia Modelu i Widoku – nie ma potrzeby, aby klasy z modelu wysyłyły komunikaty do widoku w odpowiedzi na zmianę wartości danych (w praktyce rozdzielenie Modelu i Widoku polega to na tym, że obiekty z modelu nie znajdują obiektów odpowiedzialnych za tworzenie interfejsu np. obiektów w Java Swing, C++ QT itp.)

10

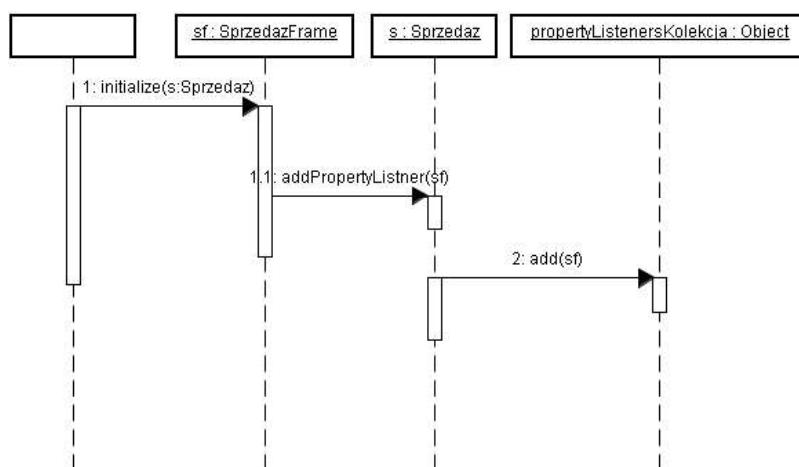
Observer: przykład (1)

- W programie wielozadaniowy POS wprowadzamy interfejs *PropertyListner*, który będzie wykorzystany do aktualizacji okien wyświetlających dane o wartości sprzedaży razem.



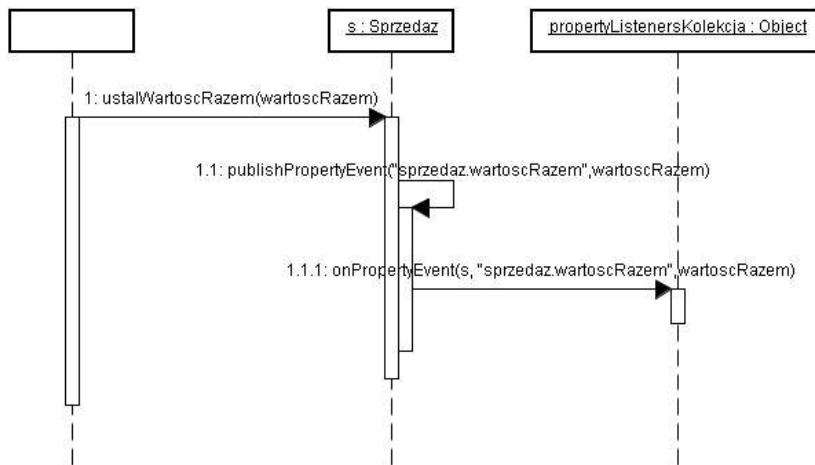
Observer: przykład (2)

- Po dodaniu obiektu słuchacza do publikatora obiekt *Sprzedaz* nie zna obiektu *SprzedazFrame*. Zachowany jest wzorzec Low Coupling.



Observer: przykład (3)

- W sytuacji, w której w obiekcie *Sprzedaz* następuje zmiana powiadamiane są wszystkie obiekty słuchacze.



13

Observer: dyskusja

- Wzorzec projektowy Observer nosił również nazwę „publish-subscribe” i wywodzi się języka Smalltalk.
- Wzorzec ten znajduje szerokie zastosowanie w technologiach takich jak Java AWT, Java Swing, czy Microsoft .NET.
- Model ten może być wykorzystany nie tylko do obsługi komunikacji z klasami GUI, ale również do modelowania wszystkich innych zagadnień w których konieczna jest komunikacja pomiędzy obiektami publikatorami, a słuchaczami.

14

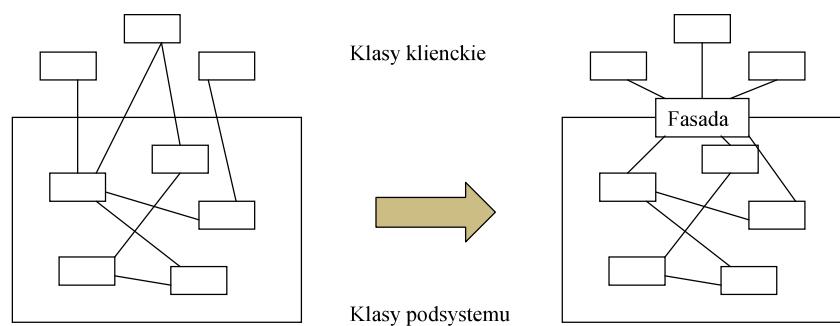
Fasada: założenia (1)

- **Problem:** W jaki sposób udostępnić jednorodny interfejs do zbioru interfejsów udostępnianych przez złożony podsystem?
- **Rozwiązanie:** Wprowadzić jeden punkt dostępowy do podsystemu – fasadę, czyli obiekt który ukrywał będzie podsystem. Obiekt ten oferuje jeden spójny interfejs oraz jest odpowiedzialny za współpracę z obiektami składającymi się na podsystem.
- Fasada definiuje interfejs wyższego poziomu, który sprawia że korzystanie z podsystemu jest prostsze.
- Fasada jest jedynym punktem dostępowym do podsystemu! Komponenty podsystemu są prywatne i nie mogą być udostępnione komponentom zewnętrznym.

15

Fasada: założenia (2)

- Fasada redukuje złożoność komunikacyjną pomiędzy podsystemami:



16

Fasada: przykład (1)

- W systemie POS konieczna jest obsługa sytuacji, w których niektóre mechanizmy systemu zachowują się nieznacznie różnie od standardowego zachowania.
- Np. nietypowo może się zachowywać metoda *utwórzElementPozycjiSprzedazy* w klasie *Sprzedaż*, tak aby blokowana była możliwość wielokrotnego jej wywołania – dotyczy sytuacji realizacji płatności przez klienta przy pomocy specjalnych bonów zezwalających na zakup jedynie produktów określonego rodzaju.
- Aby problem obsługi poprawnie podczas tworzenia klasy *Sprzedaż* przeznaczonej do obsługi danej transakcji zostanie ona oznaczona jako wymagająca innego sposobu przetwarzania – dopuszczalne jest sprzedanie jedynie jednego produktu.
- Reguły rozstrzygające, czy produkt, który ma być w danej chwili zakupiony jest akceptowany do sprzedaży za okazany bon, czy też nie będą zapisane w dedykowanym podsystemie.

17

Fasada: przykład (2)

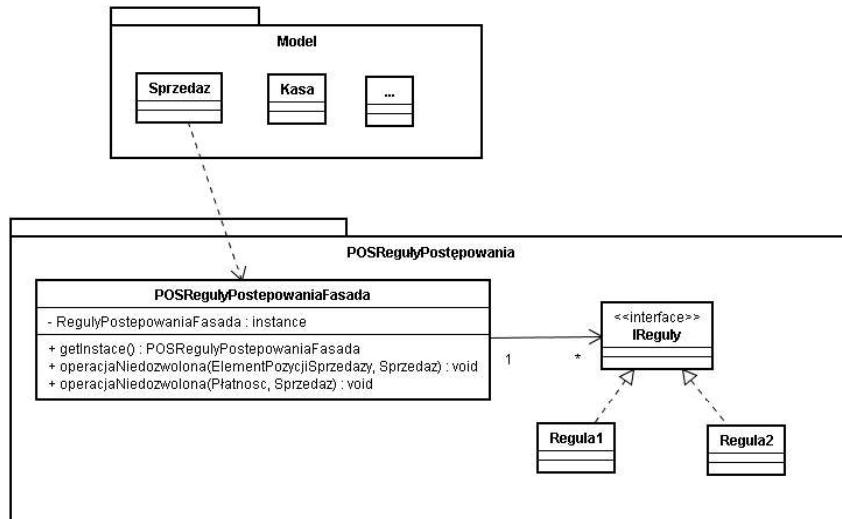
- Dostęp do podsystemu realizowany będzie poprzez fasadę *POSRegułyPostępowaniaFasada*
- Odwołanie się do fasady zapisane zostanie w metodzie, której dotyczy wariantowość:

```
public class Sprzedaż {  
    ElementPozycjiSprzedazy eps = new ElementPozycjiSprzedazy(opis, ilosc);  
    if (POSRegułyPostępowaniaFasada.getInstance.operacjaNiedozwolona(eps, this) )  
        return;  
    elementPozycjiSprzedazyKolekcja(eps);  
}  
...
```
- Rozwiążanie umożliwia ukrycie złożoności reguł postępowania w podsystemie (być może bardzo złożonym), którego zasada działania nie musi być znana podczas tworzenia danej metody.
- Rozwiążanie przedstawione powyżej wykorzystuje również wzorzec Singleton.

18

Fasada: przykład (3)

- W UML podsystemy opisane są przy pomocy elementu *package*.



19

Fasada: dyskusja

- Wzorzec fasada powinien być wykorzystany gdy:
 - Chcemy udostępnić prosty interfejs do skomplikowanego podsystemu. W takim rozwiążaniu fasada oferuje domyślny interfejs do podsystemu (w większości wypadków oznacza to również prosty interfejs). Jedynie klienci wymagający bardziej złożonych operacji będą korzystali wprost z podsystemu omijając fasadę.
 - Chcemy podzielić złożony system na logicznie różne części – fasady będą punktami dostępowymi dającymi nam takie abstrakcje.
 - Chcemy podzielić dany podsystem na warstwy. Fasady będą definiowały punkty wejścia do każdej warstwy podsystemu.
- Wzorzec Adapter jest Fasadą, która ukrywa zewnętrzne systemy różniące się interfejsami – wzorzec Fasada jest o wiele bardziej ogólny.

20