Dr. Dragan Milićev

Assoc. Professor, University of Belgrade

dmilicev@etf.rs, www.rcub.bg.ac.rs/~dmilicev

Object-Oriented Technology

Tutorial

On object-oriented concepts, languages, and patterns for engineers

Java Design Patterns

Outline

Part I:	Introduction
Part II:	Concepts
Part III:	Design Patterns
Part IV:	Conclusions

Part I: Introduction

About this Tutorial

Introduction to OO Technology

Introduction to Modeling

Chapter 1: About this Tutorial

- Subject
- Objectives
- Prerequisites
- Resources

Subject

- Fundamental OO concepts
- Basic principles of OO software design
- OO languages, techniques, and tools
  - OO programming languages (C++ and Java)
  - OO modeling language UML
  - Design patterns
- OO programming paradigm and its application to engineering domains

Objectives

- Get familiar with the basic concepts and principles of the OO paradigm
- Get introduced to the most popular OO programming and modeling languages
- Get convinced in benefits of using OO technology
- Get ready to understand the design of complex OO software systems

Prerequisites

- Experience in developing software in some engineering domains:
  - Modeling, simulation, and optimization
  - Embedded and real-time systems
  - Domain-specific modeling languages, computer-aided design tools
- Understanding of fundamental concepts of the procedural programming paradigm:
  - Type and variable
  - Declaration, expression, statement, condition, loop
  - Subprogram (procedure and function), argument (formal and actual), invocation, recursion

Resources

- Books on OO programming languages C++ and Java (many good available)
- Books on UML, OO modeling, and design patterns:
  - G. Booch, J. Rumbaugh, I. Jacobson, “The Unified Modeling Language User Guide”, Addison-Wesley, 1999
  - E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns”, Addison-Wesley, 1995
- Discussion with Dr. Milićev:
  dmilicev@rcub.bg.ac.yu
  www.rcub.bg.ac.yu/~dmilicev
- Other books on OO technology, programming languages, UML, and design patterns

Chapter 2: Introduction to OO Technology

Why OO Technology?

What Makes OO Technology?

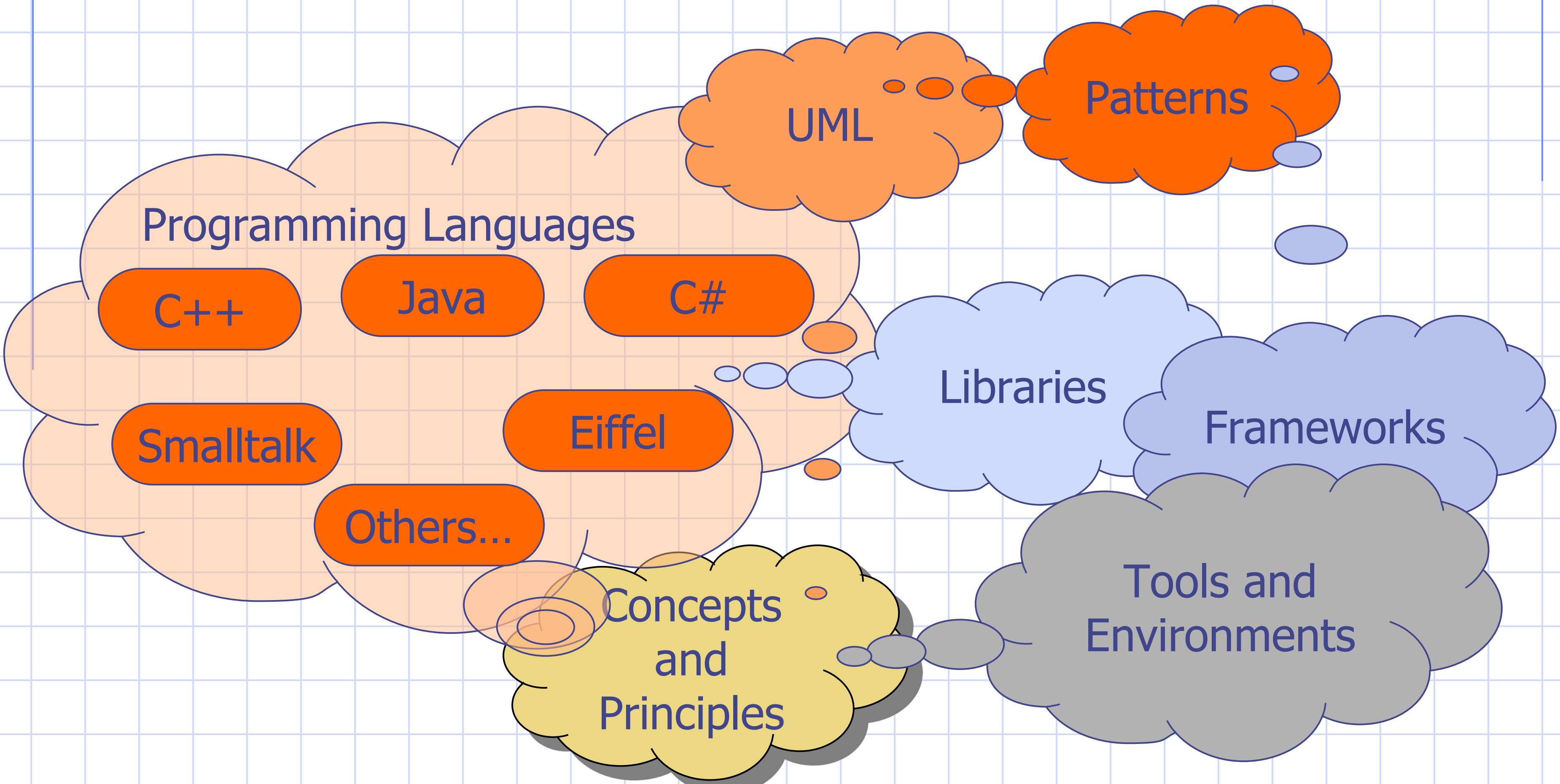
Why OO Technology?

- The most demanding problems of software development:
  - Complexity: all non-trivial software systems nowadays are complex; the users keep requiring more ambitious features; the development is long and costly
  - Maintenance: the cost of error correction and responding to modified or extended requirements is long, risky, and costly
- The traditional (procedural) paradigm could not meet these needs successfully enough!

Why OO Technology?

- How to cope with complexity, ever increasing users' needs, and demanding software maintenance?
  - Abstraction: use of highly abstract concepts, conceptually close to the problem domain, and with a lot of implicit executable semantics
  - Decomposition: clear separation of concerns, cohesive and loosely coupled modules with weak and well-controlled interfaces, localized design decisions, clear and stable software architecture
  - Intensive software reuse at various levels of granularity: code excerpts, idioms, templates, libraries, patterns, frameworks, designs, artifacts, …
- These are prerequisites for improved development productivity and less risky maintenance
- OO technology tries to overcome the drawbacks of more traditional approaches to meet these goals

What Makes OO Technology?



Object orientation is a different way of thinking about designing software!

Chapter 3: Introduction to Modeling

Models and Modeling

Sample Application

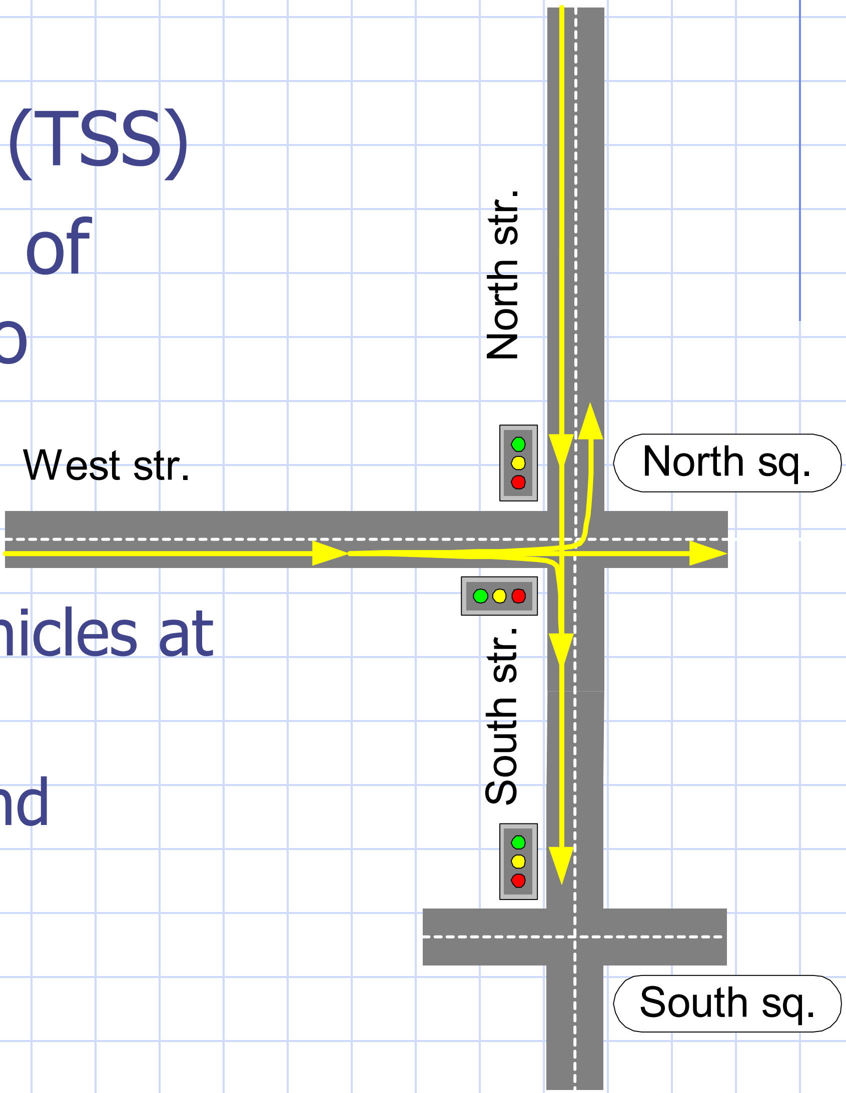
Abstractions and Conceptualization

Models and Modeling

- A model is a simplification of reality
- We build models so that we can better understand the system we are developing
- We build models of complex systems because we cannot comprehend such a system in its entirety
- Models:
  - help us to visualize a system as it is or as we want it to be
  - permit us to specify the structure or behavior of a system
  - give us a template that guides us in constructing a system
  - document the decisions we have made
- Modeling is a central part of all the activities that lead up to the deployment of good software

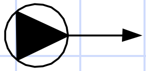
Sample Application

- Traffic Simulation System (TSS)
- Task: find a proper timing of traffic lights, so there is no congestion at crossroads
- Requirements:
  - Random occurrences of vehicles at streets
  - Vehicles pass the streets and squares with random delay
  - Vehicles turn left and right randomly

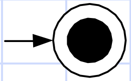


Abstraction and Conceptualization

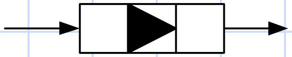
- An abstraction is the essential characteristics of an entity that distinguish it from all other kinds of entities. It is a simplified representation of an entity from the problem domain
- Proposed abstractions in TSS:



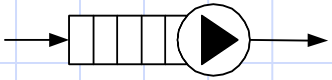
Flow Source: generates a random traffic flow at street entrances



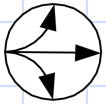
Flow Sink: sinks the input traffic flow at streets that are of no interest



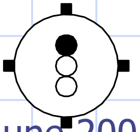
Pipe: models a random delay of traffic through a street



Queued Server: models a line at a traffic light and passage through a crossroad



Switch 1 to 3: models random turns of vehicles



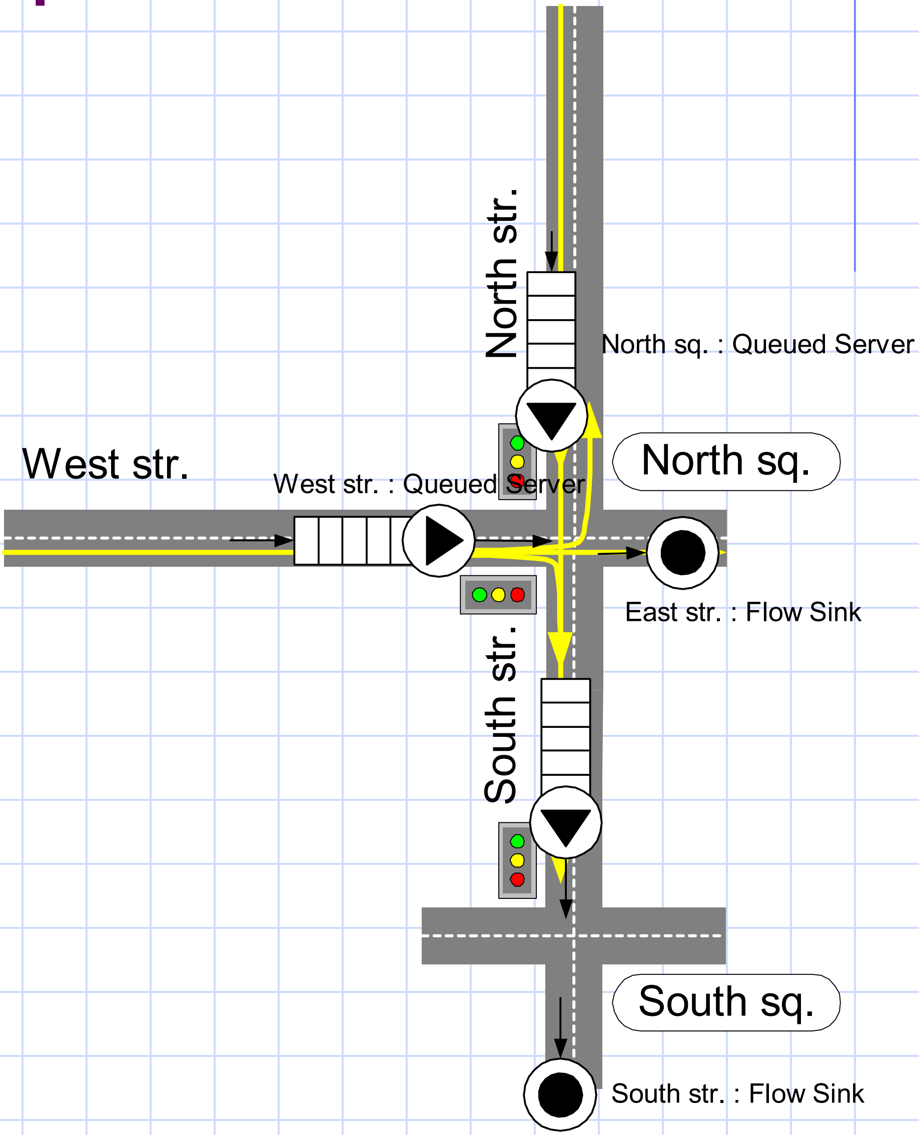
Cross Semaphore: models a traffic light with two controlled directions

Abstraction and Conceptualization

- An abstraction defines a boundary relative to the perspective of the viewer
- An abstraction neglects some differences from entities in real world, in favor of generalizing their commonalities
- Abstractions are characterized with their *semantics*, *properties*, *relationships*, and *behavior*
- A *key abstraction* is an abstraction that is important enough to be incorporated into the system's conceptual model regardless to its concrete implementation

Abstraction and Conceptualization

- An abstraction defines a set of *instances*. For example, North sq., South sq., or West sq. are all instances of the *queued server* key abstraction. Similarly, East str. and South str. are instances of the *flow sink* abstraction

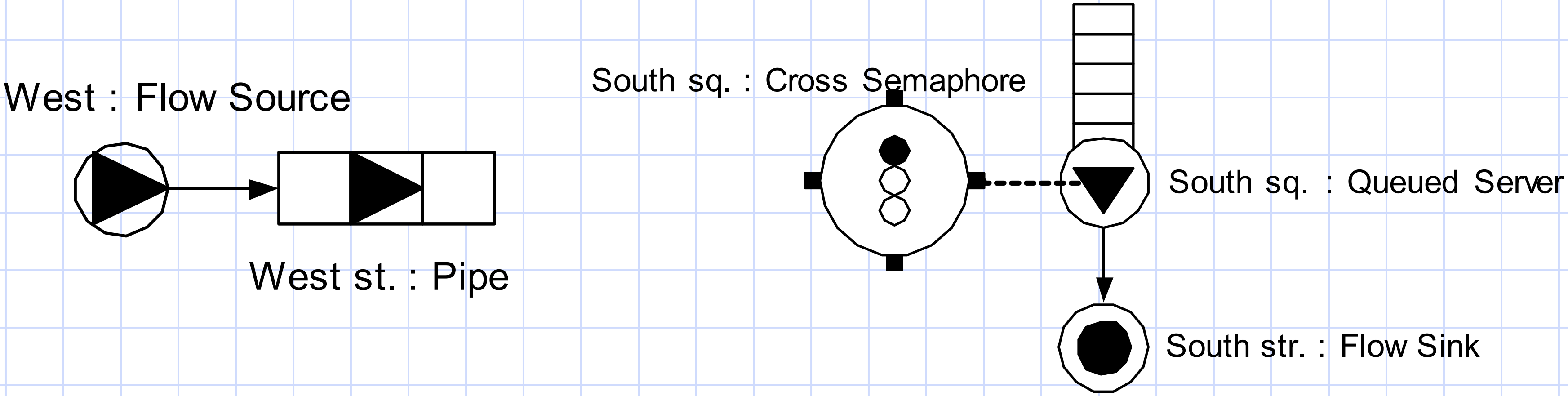


Abstraction and Conceptualization

- An abstraction is characterized with its *properties*. For example, the properties for the *flow source* abstraction are name, description, and mean frequency of vehicle occurrences, and for the *pipe* abstraction are name, description, and mean delay of vehicles, etc.
- Each instance of an abstraction will have its own value of each property.
  For example, the West str. flow source will have the number 1/5 as the value of its mean frequency property, and the North str. pipe will have 4 as the value of its mean delay property

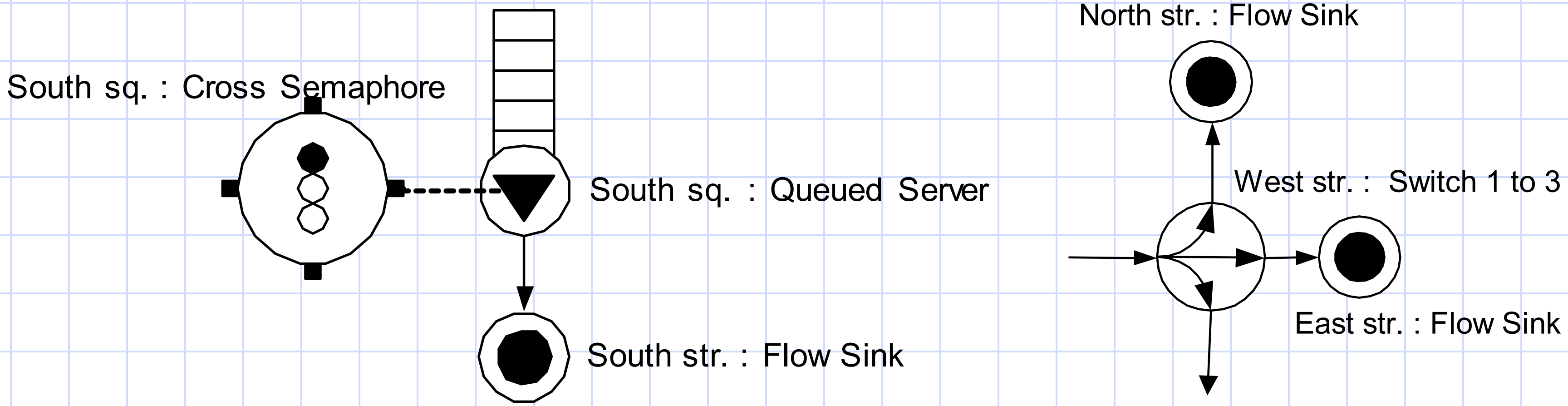
Abstraction and Conceptualization

- Abstractions have their relationships.
  For example, the *flow target* of the West str. flow source is the West str. Pipe, and the South sq. cross semaphore *controls* the South sq. queued server



Abstraction and Conceptualization

- Abstractions have their *behavior*, meaning that their instances react on certain stimuli.
  For example, a queued server stops serving vehicle occurrences when it is closed by a cross semaphore, or a switch transfers a vehicle occurrence to one of its targets randomly


South sq. : Cross Semaphore
South sq. : Queued Server
South str. : Flow Sink
North str. : Flow Sink
West str. : Switch 1 to 3
East str. : Flow Sink

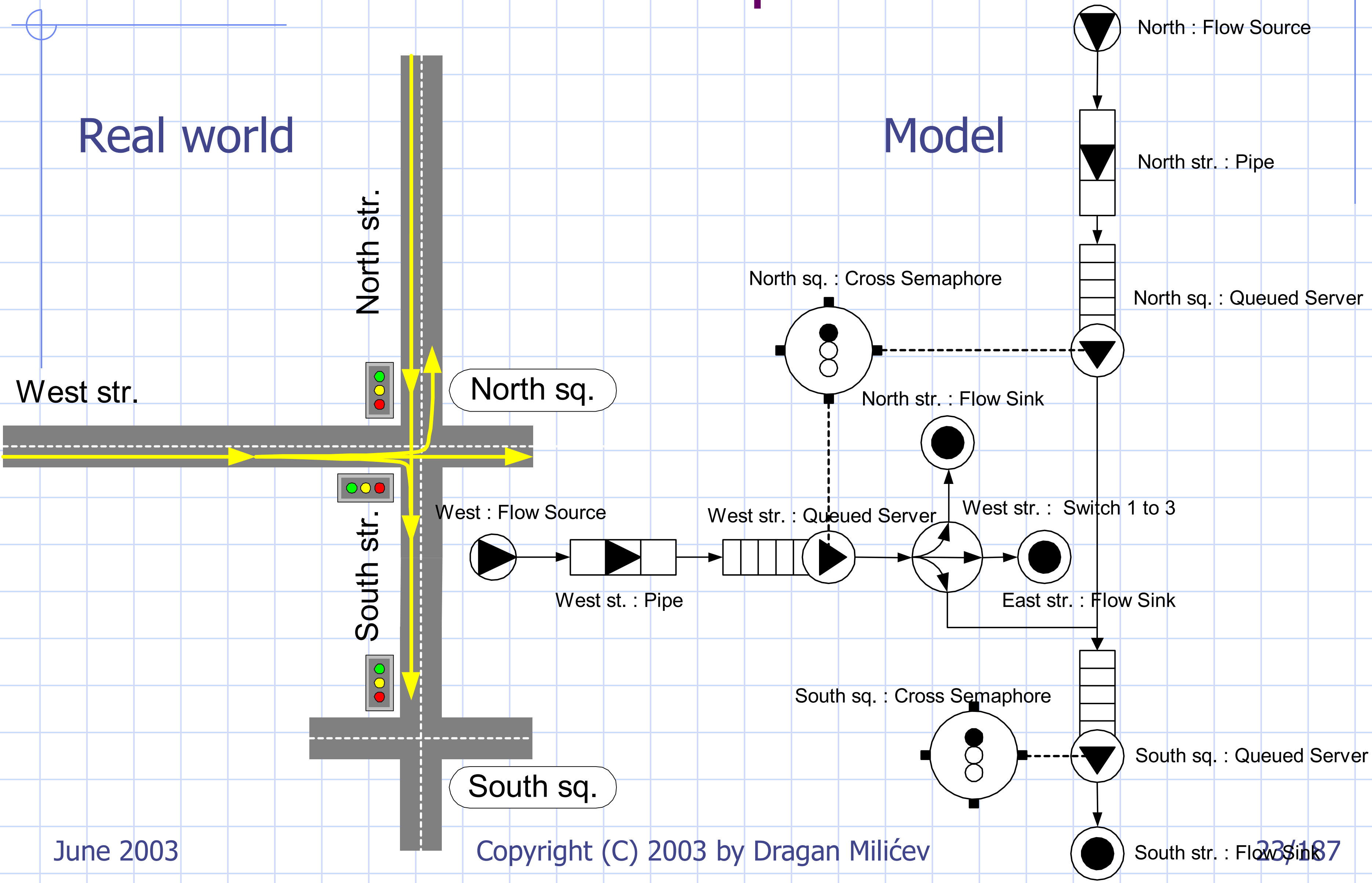
Abstraction and Conceptualization

- The process of discovering and inventing the proper abstractions is called *conceptualization*
- The model that comprises of key abstractions and their properties and relationships is called the *conceptual model* of the problem domain
- OO modeling is the process of modeling using OO concepts
- Principles of OO decomposition drive this process:
  - Proper separation of concerns and assignment of responsibilities
  - Generalization/specialization hierarchies
  - Designing interactions among objects
  - Programming by interfaces
- OO design patterns help us to find elegant solutions to frequent problems in different contexts

Abstraction and Conceptualization

Real world

Model



Part II: Concepts

- Classes
- Attributes
- Structural Relationships
- Generalization/Specialization
- Operations
- Polymorphism
- Encapsulation
- Interfaces
- Interactions

Chapter 4: Classes

Motivation

Concepts

Support in UML

Support in C++

Support in Java

Advanced Concepts

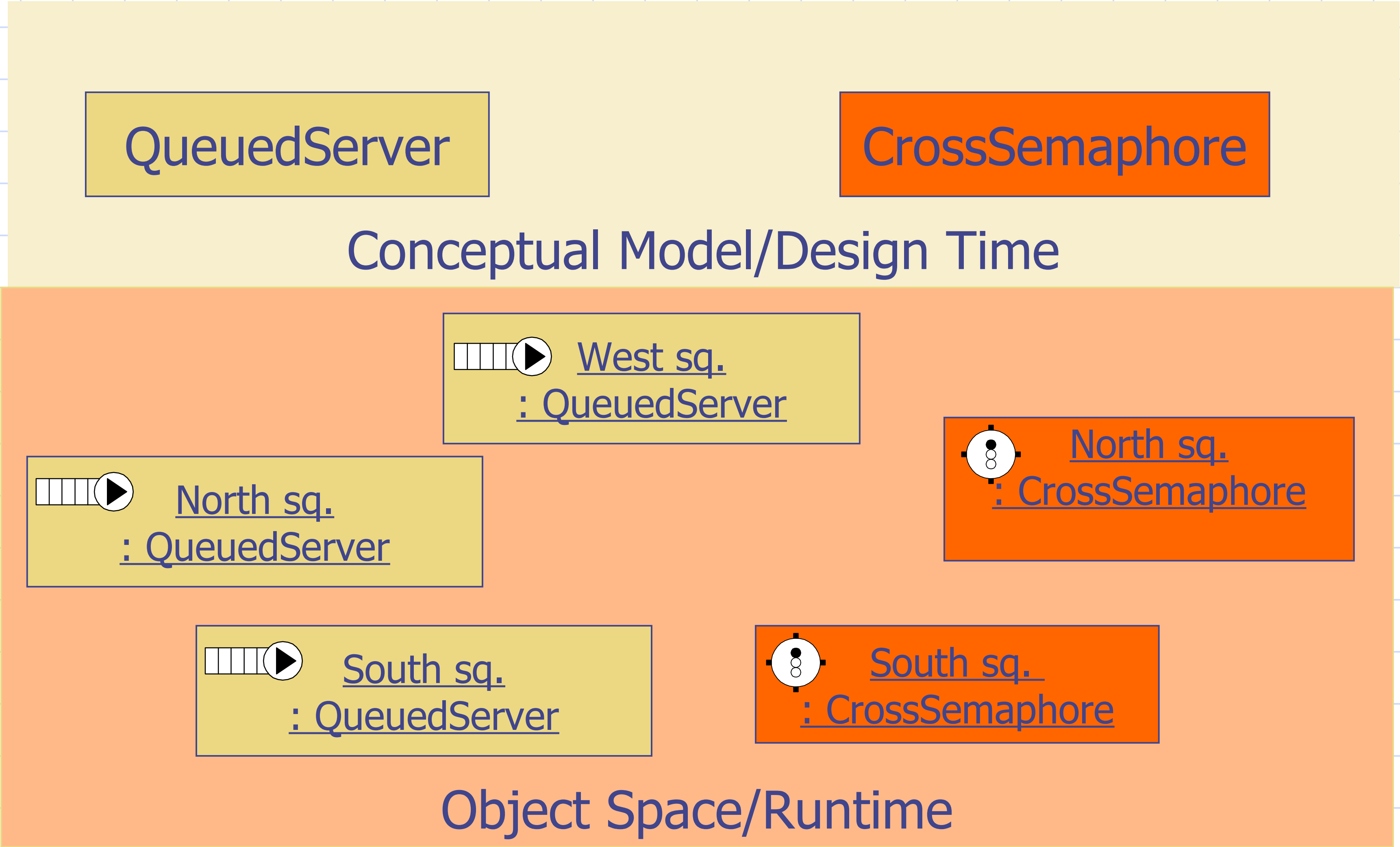
Motivation

- In TSS, we have identified the key abstractions, like *flow source, pipe, queued server*, etc. They represent sets of instances that exist in the system at runtime.
- The instances from the same set (of the same abstraction) share the common semantics, structure, relationships, and behavior.

Concepts

- In OO software engineering, abstractions are modeled with *classes*
- *Class* is a description of a set of objects that share the same properties, behavior, relationships, and semantics
- *Object* is an instance of a class - a concrete manifestation of an abstraction. It is an entity with well-defined boundary and *identity*, which encapsulates *state* and *behavior*
- **`FlowSink`**, **`CrossSemaphore`**, etc. are classes in the model, and “South str.” and “North sq.” are objects of these classes in the executing system

Concepts



QueuedServer

CrossSemaphore

Conceptual Model/Design Time

West sq.
: QueuedServer

North sq.
: QueuedServer

North sq.
: CrossSemaphore

South sq.
: QueuedServer

South sq.
: CrossSemaphore

Object Space/Runtime

Concepts

- A class is an element of the conceptual model of a domain
- Classes reside:
  - in the conceptual space (model)
  - at design time
- Classes describe sets of objects that share the same structure and behavior. A class describes that structure and behavior, and represents a template for creating objects

Concepts

- Objects are instances of classes
- Objects:
  - reside in the system's object space
  - live at runtime
- Each object has its:
  - identity
  - type
  - internal state that is defined by the dynamic value of the structure defined in its class
  - capability to provide behavior as defined in its class, when this is requested from it
- The set of all objects of one class in the system's object space is called the *extent* of that class

Concepts

- A class and its objects are related by the type-instance dichotomy (objects are instances of classes)
- A class is an abstract, conceptual thing. Objects are concrete, physical things
- A class is only a description of a set of objects, i.e., the structure and behavior that objects share
- Classes live in the developers' and users' minds. Objects live in time and space
- Classes reside at the design side, and objects at the execution side of the dichotomy

Concepts

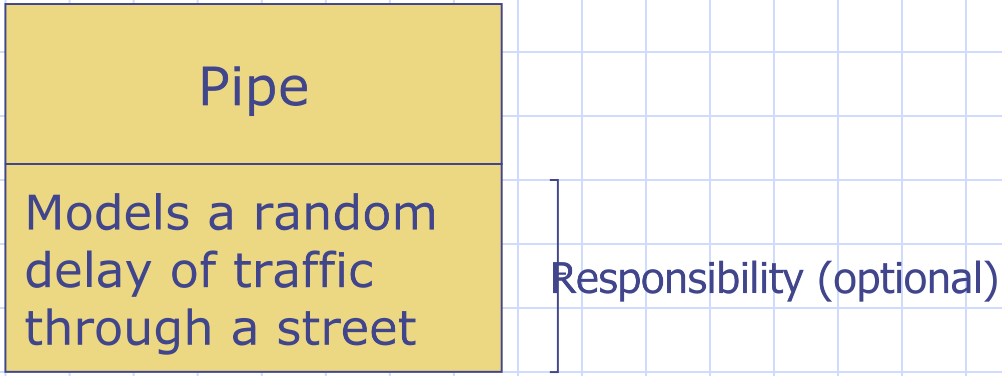
- When a class is defined in the conceptual model, it has the following semantics:
  - Objects of that class can exist in the system's object space at runtime. These objects can be *created* and *destroyed*
  - These objects will share the same structure, behavior, and relationships
  - The objects will have the same semantics, i.e., they represent instances of an abstraction that has a particular important meaning in the domain's conceptual space

Concepts

- An object has the following characteristics:
    - An object is an instance of its class. An object of a class can be distinguished from all other objects of the same and other classes
    - An object has its lifetime, meaning that it lives since its *creation*, until its *destruction*. An object can be accessed only during its lifetime. Besides, an object takes some space in the computer system's memory. Therefore, objects live in time and space

Support in UML

- In UML, an object can be an instance of several classes at the same time
- Notation for a class:

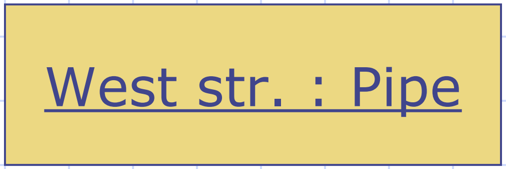


Pipe

Models a random delay of traffic through a street

Responsibility (optional)

- Notation for an element of the model that represents a particular or prototype object:



West str. : Pipe

Support in UML

- The *action semantics* of UML defines two basic actions:
  - Create a new object of a certain class
  - Destroy an object
- The notation for these actions is not specified, but it is left to the implementations to define the *surface language* for specifying actions in the program

Support in C++

- Definition of a class:

```
class Pipe {
  //...Definition of structure and behavior
};
```

- Creation of a new object (the operator **new** returns a pointer to the created object):

```
new Pipe
```

- Destruction of an object over a pointer to it:

```
Pipe* aPipe = new Pipe; // aPipe is a pointer
                        // to the created object
//... Usage of the object over the pointer aPipe
delete aPipe; // Destroy the object over the pointer
```

- In C++, there are other types of lifetimes of objects with implicit construction and destruction

Support in Java

- Definition of a class:

```
class Pipe {
  //...Definition of structure and behavior
}
```

- Creation of a new object (the operator **`new`** returns a reference to the created object):

```
Pipe aPipe = new Pipe; // aPipe is a reference
                       // to the created object
```

- In Java, an object is destroyed implicitly by the built-in garbage collector, once there are no more references linked to it

Advanced Concepts

- A class is *persistent* if its objects can survive the termination of the execution context of their creation. Examples:
  - A TSS model should be stored into a file to be restored for a new execution of the simulation application
  - Database applications
- A class is *active* if its objects encapsulate separate threads of control (processes). For example, the elements of a TSS model (e.g. flow sources, semaphores, etc.) can be implemented by concurrent threads in the underlying runtime environment
- C++ and Java do not support persistency explicitly (objects are stored in volatile computer memory). Java supports active classes, but C++ does not

Advanced Concepts

- An *abstract data type* is a set of instances that share the same semantics, properties, relationships, and behavior, but that are pure values that do not have their identities, meaning that two instances with the same values cannot be distinguished
- Instances of data types are predominantly used as attribute values of objects or arguments of operations that may be arbitrarily copied (they usually have meaningful copy semantics)
- Examples: Integer, Complex, String, Date, Time, Currency, etc.
- C++ and Java do not distinguish classes and data types, while UML does

Chapter 5: Attributes

Motivation

Concepts

Support in UML

Support in C++

Support in Java

Advanced Concepts

Motivation

- In TSS, although instances of the same class share the same set of properties, each of it has a different value of a property.
- For example, the properties for the *flow source* abstraction are name, description, and mean frequency of vehicle occurrences, and for the *pipe* abstraction are name, description, and mean delay of vehicles, etc.
- For example, the West str. flow source will have the number 1/5 as the value of its mean frequency property, and the North str. pipe will have 4 as the value of its mean delay property.

Concepts

- Properties of abstractions are modeled with *attributes*
- *Attribute* is a named property of a class that describes a range of values that instances of the property can hold
- An attribute is a member of a class
- For example, the attributes of the class **`Pipe`** may be:
  - name: represents a short title of the element
  - description: allows a longer textual description of the element
  - mean delay: mean value of a random delay of vehicles through the pipe

Concepts

- An attribute is defined by its:
  - name: a string that uniquely identifies the attribute in the scope of its class (e.g. **`name`**, **`descr`**, and **`meanDelay`** are names of the attributes of the **`Pipe`** class)
  - type: defines the range of values that the instances of that property can hold, along with the operations applicable on these values (e.g. the type of the **`meanDelay`** attribute is **`Real`**)
  - default value: the initial value that an instance of the attribute has when an object is created (e.g. the attribute **`meanDelay`** of the class **`Pipe`** can have the default value set to zero)
- The values of the attributes can be accessed and modified according to the specification of their types

Concepts

- Attribute is a conceptual thing and lives at the “design” side of the dichotomy. It is a member of a class
- Objects have values of the attributes
- The values of the attributes are instances of these attributes. Each object of the class has its own value of the attribute, which is independent of the other objects' values
- In some languages, the lifetime of attribute instances are tied to the lifetimes of their enclosing objects

Concepts

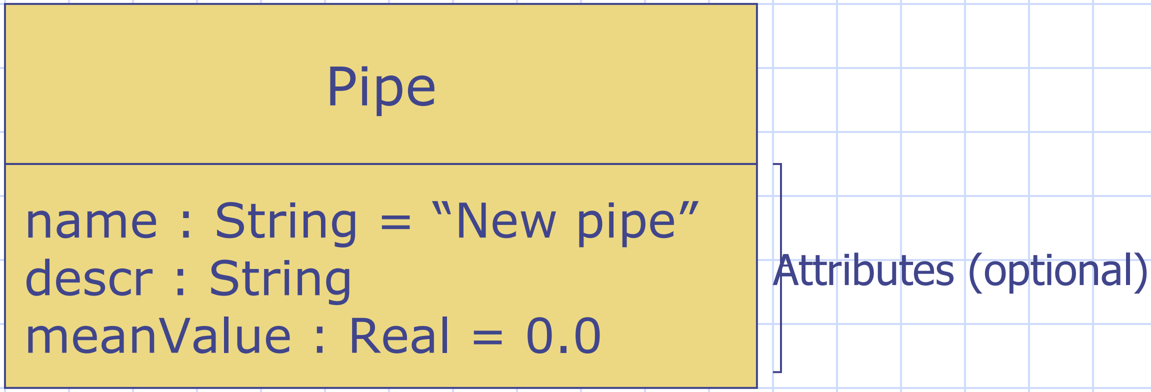
- Objects change their states, which are defined by the values of their attributes, throughout their lifetimes
- The attributes define a static structure of the objects of a class, which is shared by these objects, and the attribute values define the current states of the objects, which are proprietary to the objects

Support in UML

- In UML, the type of attributes can be a class or a data type
- In UML, an attribute may have multiple values, with the specified cardinality
- UML does not specify the dependency of the lifetime of attribute values on the lifetime of the enclosing object
- In UML, an attribute is a reference to an instance of a class or a data type. The reference's lifetime is bound to the enclosing object's lifetime, but the lifetime of the referenced instance is not

Support in UML

- Notation for attributes:

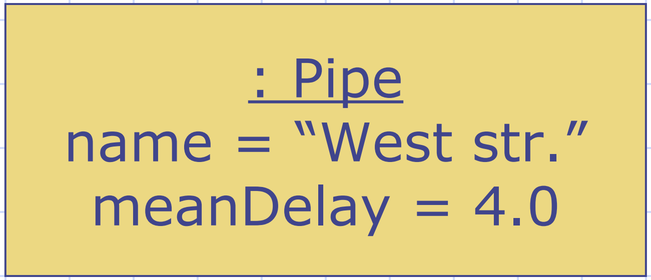


Pipe

name : String = “New pipe”
descr : String
meanValue : Real = 0.0

Attributes (optional)

- Notation for an element of the model that represents a particular or prototype object at runtime, with a specific set of attribute values:



: Pipe
name = “West str.”
meanDelay = 4.0

Support in UML

- The action semantics of UML defines two basic actions:
  - Read the value of an attribute of a certain object
  - Write the given value of an attribute of a certain object
- The notation for these actions is not specified, but it is left to the implementations to define the surface language for specifying actions in the program

Support in C++

- In C++, attributes can be instances of:
  - Built-in data types (e.g., int, char, float, etc.), including pointers/references to other objects
  - Classes (object incorporated *by value*)
- The lifetime of an attribute value is bound to the lifetime of the enclosing object: the attribute value is created and destroyed along with the object
- However, if the attribute is a pointer/reference to another object, the destruction is *not* implicitly propagated to the referred object
- Default (initial) values of attributes are supported indirectly (through constructors, to be explained later)

Support in C++

- Specification of attributes embodied by value (their lifetime is bound to the enclosing object):

```
class Pipe {
  String name; // Attribute embodied by value
  String descr;
  double meanDelay;
};
```

- Specification of attributes embodied by pointer/reference (their lifetime is not implicitly bound to the enclosing object):

```
class Pipe {
  String* name; // Attribute embodied by pointer.
                // No initial value for the pointer
    ...         // is assumed
  };
```

Support in C++

- Access to attribute values:
  - If the attribute is embodied by value:

```
Pipe* aPipe = new Pipe;
aPipe->meanDelay = 4.0; // Write attribute value
double temp = aPipe->meanDelay; // Read attr. value
// Access the embodied object of the class String:
aPipe->name.setValue("West str.");
...
delete aPipe; // Destroy the object
```

  - If the attribute is embodied by pointer:

```
Pipe* aPipe = new Pipe;
// Create the pointed object of the class String:
aPipe->name = new String;
// Access the pointed object over the pointer attr.:
aPipe->name->setValue("West str.");
...
delete aPipe->name; // Destroy the pointed object
delete aPipe; // Destroy the object
```

Support in Java

- In Java, attributes can be instances of built-in data types only (e.g., int, char, float, etc.), including references to other objects of classes
- The lifetime of an attribute value is bound to the lifetime of the enclosing object: the attribute value is created and destroyed along with the object
- However, if the attribute is a reference to another object, the destruction of the referenced object is determined by other references that refer to it (it is destroyed implicitly when no references refer to it any more)
- Default (initial) values of attributes are supported directly

Support in Java

- Specification of attributes:

```
class Pipe {
  // References to objects of type String:
  String name = new String(“New pipe”);
  String descr = new String;

  // Attribute of a built-in type:
  double meanDelay = 0.0;
}
```

- Access to attribute values:

```
   Pipe aPipe = new Pipe;
   Pipe.meanDelay = 4.0; // Write attribute value
   double temp = aPipe.meanDelay; // Read attr. value
   // Access the embodied object of the class String:
   aPipe.name.setValue(“West str.”);
   ...
```

Advanced Concepts

- An attribute can be of *class scope* (instead of *instance scope*), meaning that all objects of the attribute's class share the same value
- UML, C++, and Java support attributes of class scope directly (called *static* attributes in C++ and Java)
- An attribute can be *read-only*, meaning that its value cannot be modified
- UML, C++, and Java support read-only attributes (through **`const`** types in C++ and **`final`** specifier in Java)

Chapter 6: Structural Relationships

Motivation

Concepts

Support in UML

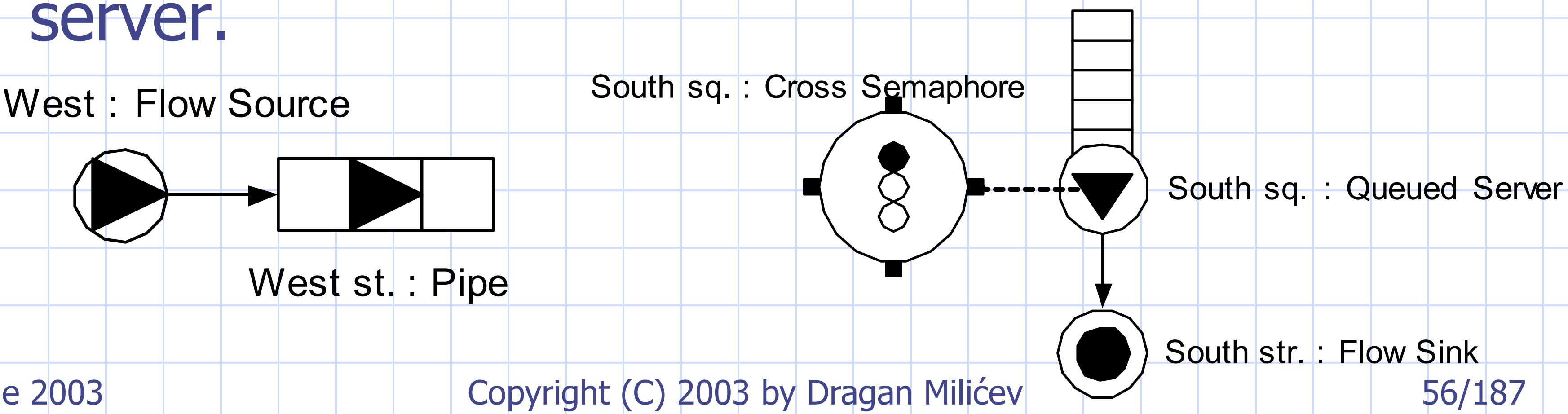
Support in C++

Support in Java

Advanced Concepts

Motivation

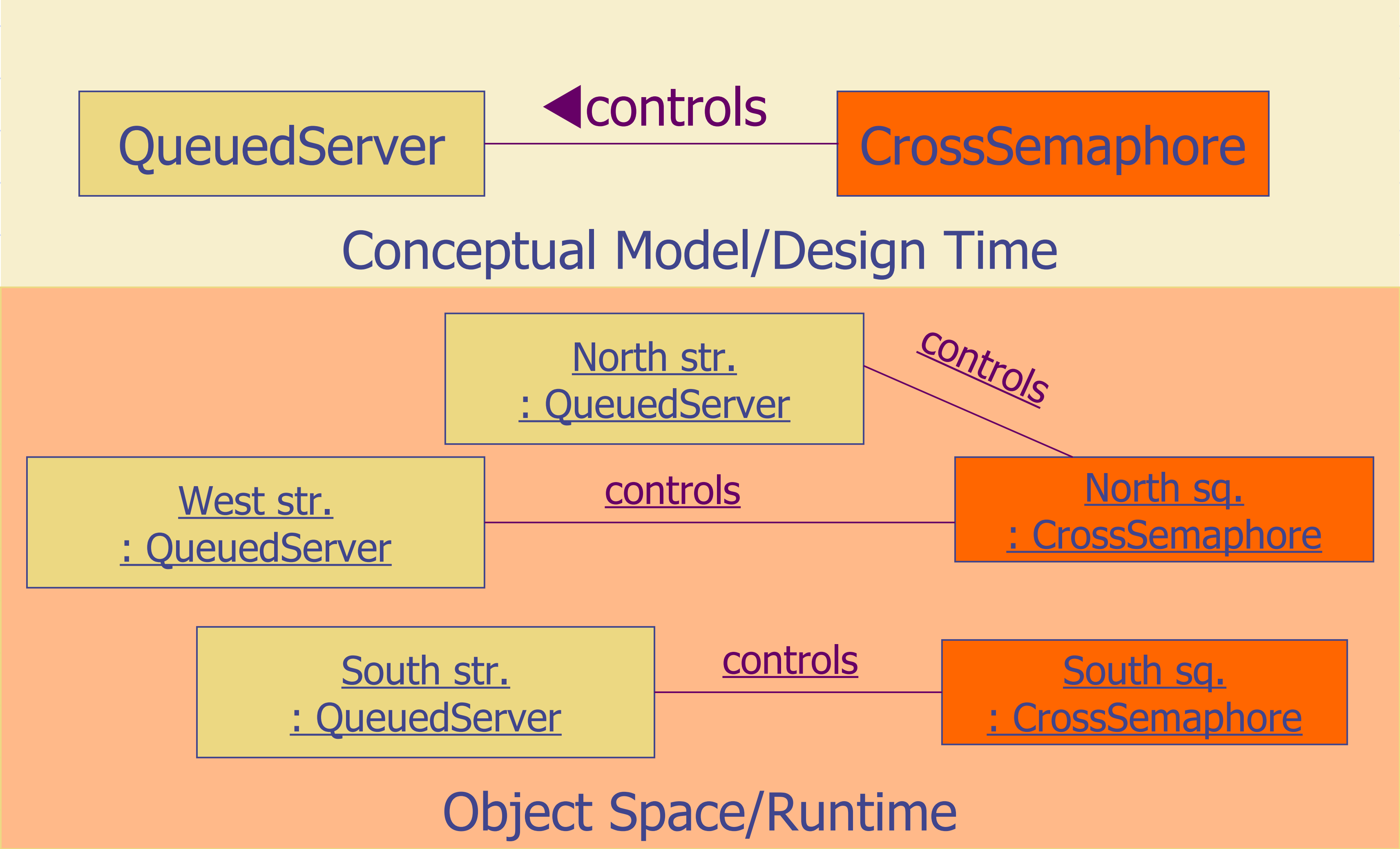
- In TSS, objects do not exist unrelated. On the contrary, they are connected to form a complex structure, whereby the interconnections conceptualize different aspects.
- For example, the *flow target* of the West str. flow source is the West str. Pipe, and the South sq. cross semaphore *controls* the South sq. queued server.



Concepts

- Different OO languages support different structural relationships, but ultimately they all allow interconnections of objects in graph-like structures, whereby the objects are interconnected by links that conceptualize relationships from the problem domain and allow navigation between objects
- *Association* is a structural relationship among classes that describes a set of *links*, in which a link is a connection among objects
- Association is a semantic relationship between two classes that involves connections among their instances

Concepts


QueuedServer
◀controls
CrossSemaphore
Conceptual Model/Design Time
North str.
: QueuedServer
controls
West str.
: QueuedServer
controls
North sq.
: CrossSemaphore
South str.
: QueuedServer
controls
South sq.
: CrossSemaphore
Object Space/Runtime

Concepts

- Links exist at runtime, in the system's object space
- Links are structural connections between objects
- A link does not have its identity – it is identified by the objects it connects
- A link does not have its independent life – a link dies when an object on any side is destroyed. In general, a link cannot exist without the objects on both sides
- In general, a link either exists or not – there can be no more than one link of the same association between the same two objects (unless the link has other specifiers)

Concepts

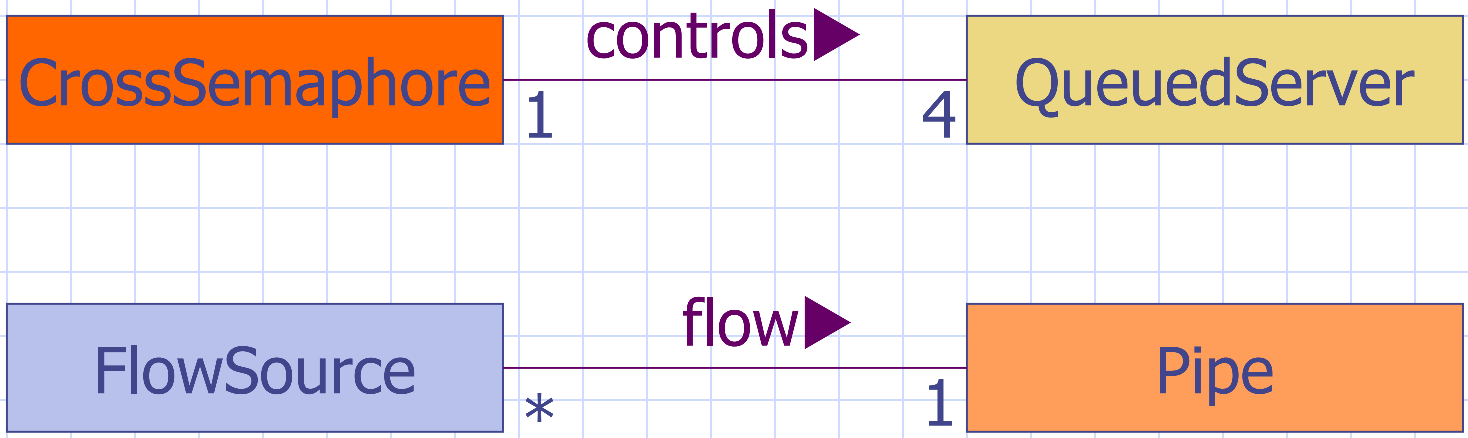
- Links are structural connections between objects, meaning that the system can navigate through the object structure, by accessing the objects linked to one object over the links of a certain association
- Links are instances of associations
- Association is a relationship between classes, while links are connections between instances of classes
- Association is a description of a set of links
- Association is a conceptual thing that exists in the conceptual model, at design time – type/design side of the dichotomy
- Links are physical things that exist in the object space, at runtime – instance/execution side of the dichotomy

Concepts

- The object structure of the system can be regarded as a typed graph, whereby objects are the nodes, and links are the edges of the graph
- The graph is typed, because each node (object) has its type – that is the class of which the object is an instance, and each edge (link) has its type – that is the association of which the link is an instance
- The system manipulates with the graph by creating and destroying the objects, reading and modifying their attribute values, creating and destroying its links, and traversing the objects over the links

Concepts

- *Multiplicity* is an adornment of an association end
- Multiplicity is a specification of the range of allowable cardinalities of the links at the opposite side of the association

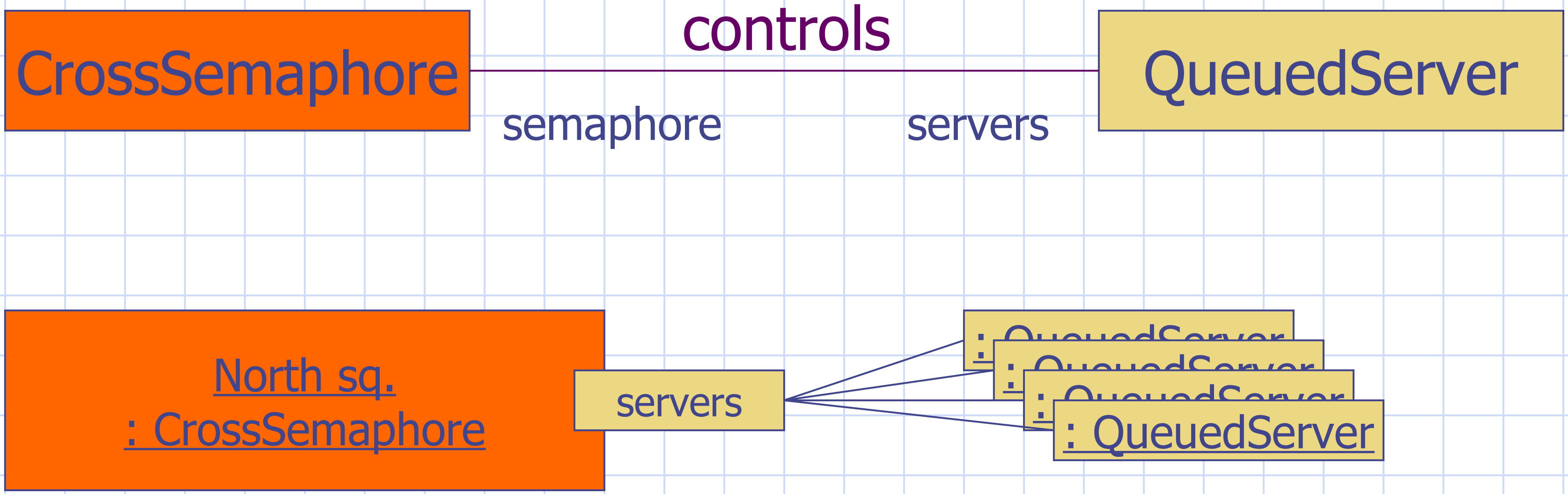


Concepts

- Multiplicity specifications can be:
  - 0..1 (zero or one)
  - 1 (exactly one)
  - * (zero to arbitrary many)
  - 1..* (one to arbitrary many)
  - a specific range *m..n* (*m* to *n*, where *n* can be * – arbitrary many)
- The multiplicity *m* at the side B of an association means that one object at the side A can be linked with *m* objects of the side B by links of this association

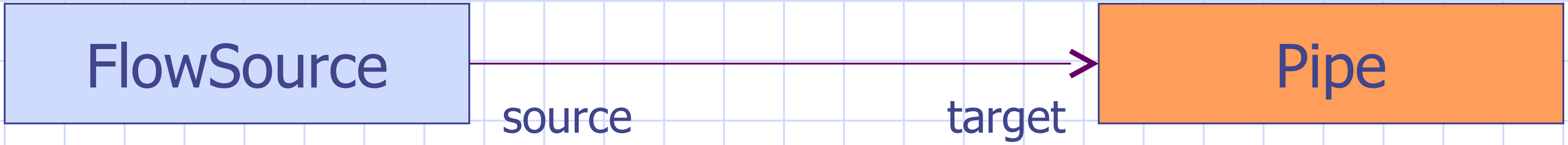
Concepts

- Each association end can be adorned with a *role*
- In all objects of the class at side *A*, there will exist a *property* named as the role at side *B*
- This property represents a “hook” on which the links of the corresponding association are “hanged,” and designates a set of linked objects



Concepts

- *Navigability* is the specification of the capability to navigate to the linked objects in the set designated by the property that is the consequence of the assoc. role
- If the association is not navigable at one side, there will be no property that results in the object set in the objects of the opposite class
- An association can be bidirectional or unidirectional (navigable at one side only)



Support in UML

- UML supports associations as described so far, with the given notation, along with some advanced concepts like:
  - N-ary association (association between many classes)
  - Association class (a class that is also an association)
  - Qualifiers of association ends
  - Other specifiers of association ends (ordered, unique, read-only, derived, subsets, unions, etc.) that specify the characteristics of the object set designated by the property
- UML action semantics define actions for manipulating links:
  - Create link of an association between a set of objects
  - Destroy a link
  - Navigate over links (access the object set of a property)
- UML 2.0 will equalize attributes (possibly multivalued) and association ends: both designate properties of objects that result in sets of instances of classes or data types

Support in C++

- C++ supports unidirectional associations through pointers to objects:

```
class FlowSource {
  ...
  Pipe* target;
};
```

- Associations with multiplicity * must be implemented by collections of pointers – data types implemented with classes (no built-in data structure for this):

```
class Model {
  ...
  CollectionOfSemaphores allSemaphores;
};
```

Support in C++

- Creation of links is done by setting the values of pointers:

```
class FlowSource {
  ...
  Pipe* target;
};
...
FlowSource* aFS = new FlowSource;
Pipe* aPipe = new Pipe;
aFS->target = aPipe;
```

- Destruction of links is done by setting the value of a pointer to null:

```
aFS->target = 0;
```

- C++ pointers are unsafe: there is no runtime checking of pointer validity (against null and dangling pointers)!

Support in Java

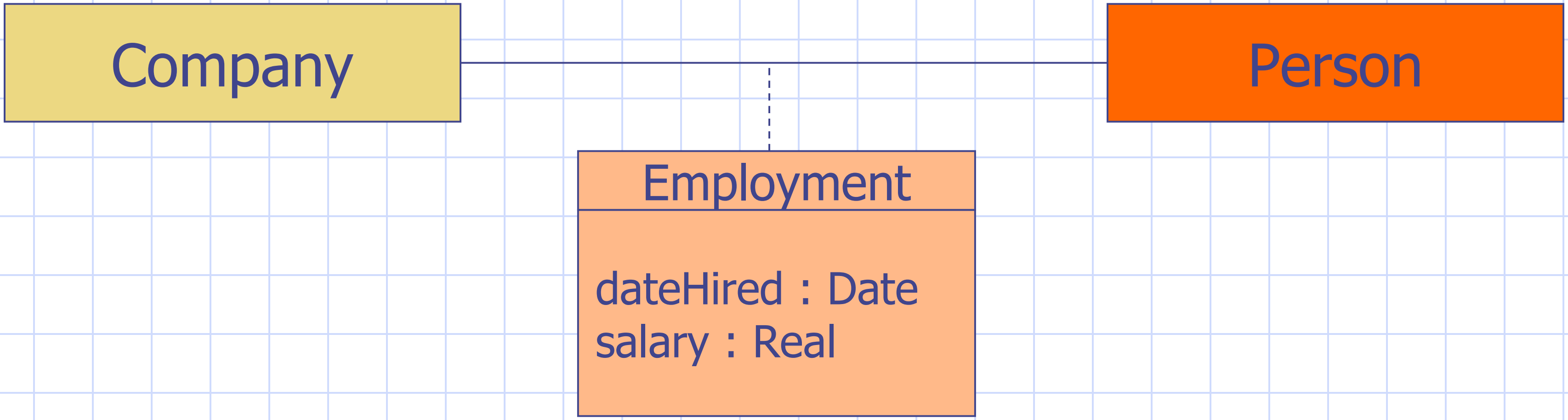
- Java supports object links in a very similar manner as C++, except for the notation (no operators needed):

```
class FlowSource {
  ...
  Pipe target;
}
...
FlowSource aFS = new FlowSource;
Pipe aPipe = new Pipe;
aFS.target = aPipe;
...
aFS.target = null;
```

- Java references are more safe than C++ pointers: there are no dangling references, because an object cannot be deleted if a reference refers to it!

Advanced Concepts

- *Association class* is an association that is also a class
- An association class can participate in relationships and have members as any other class
- Instances of association classes are link-objects
- A *link-object* is a link that is also an object
- Supported in UML, but not in C++ and Java



Chapter 7: Generalization/Specialization

Motivation

Concepts

Support in UML

Support in C++

Support in Java

Advanced Concepts

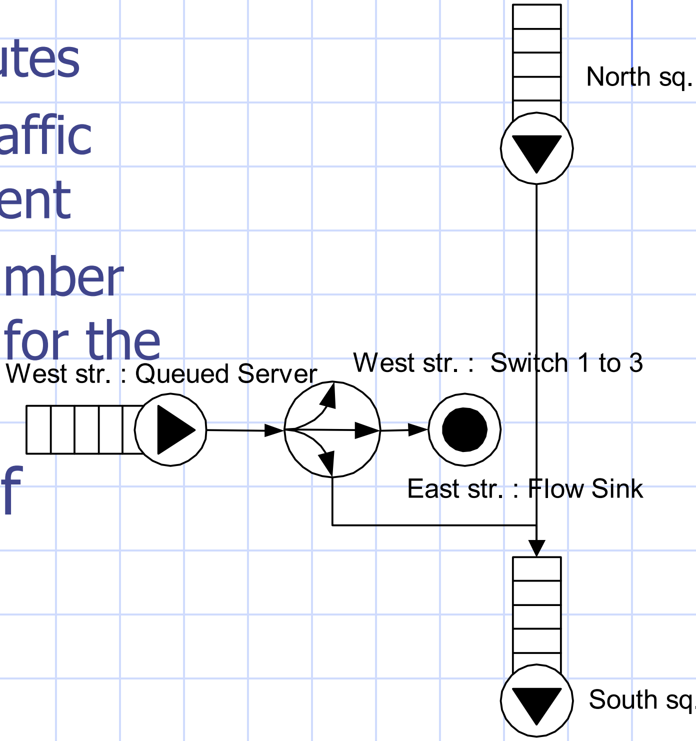
Motivation

The discovered abstractions in TSS have some things in common:

- they all have name and description attributes
- apart from semaphores, the target of a traffic flow element may be any other flow element
- a traffic flow element should count the number of vehicle occurrences it has transported, for the simulation report purposes.

Unless there is a proper generalization of these abstractions,

- the software model will contain a lot of redundancy and
- there will be no easy way to connect flow elements arbitrarily and interchangeably.



Motivation

Suppose there is a new requirement for TSS to introduce another abstraction of *adaptable cross semaphore*.

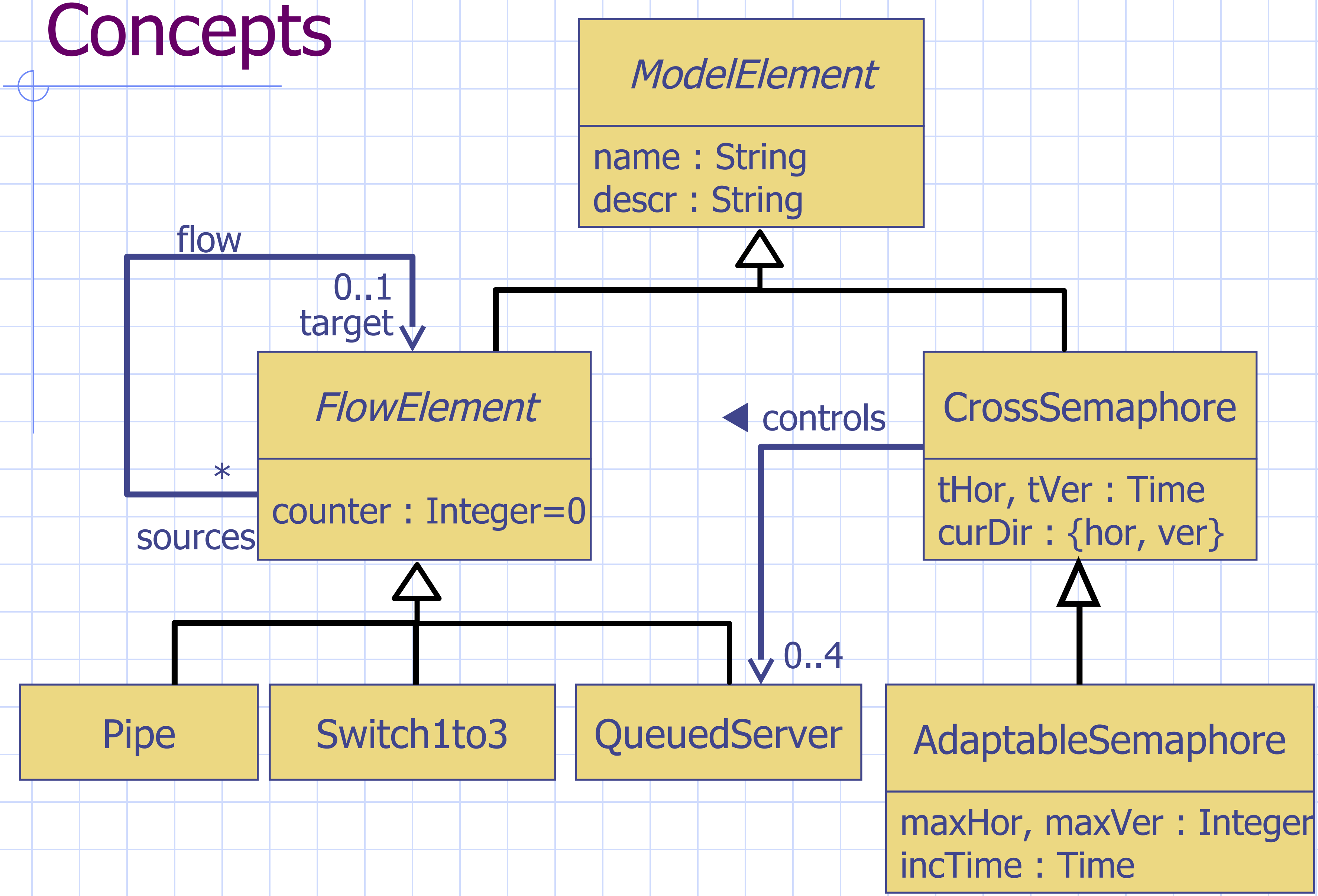
Adaptable semaphore is a kind of a cross semaphore, because it also controls two directions and opens their flow alternatively with a predefined time intervals.

However, adaptable semaphore is a special kind of a cross semaphore, because it adapts its timing according to the traffic congestion at queued servers (i.e., their maximal recorded queue size). If the maximal recorded queue size of an associated queued server is greater than a defined value, the semaphore will increment the green-light duration for the corresponding direction.

Concepts

- In OO paradigm, one of the most important kinds of relationships is the *generalization/specialization* relationship
- *Generalization/specialization* is a relationship between classes, in which objects of the specialized class are substitutable for objects of the generalized class
- The specialized class is also called the subclass, the derived class, the child, or the descendent. The generalized class is also called the superclass, the base class, the parent, or the ancestor
- This relationship is sometimes also called *inheritance*

Concepts


ModelElement
name : String
descr : String
flow
0..1
target
*
sources
FlowElement
counter : Integer=0
◀ controls
CrossSemaphore
tHor, tVer : Time
curDir : {hor, ver}
0..4
Pipe
Switch1to3
QueuedServer
AdaptableSemaphore
maxHor, maxVer : Integer
incTime : Time

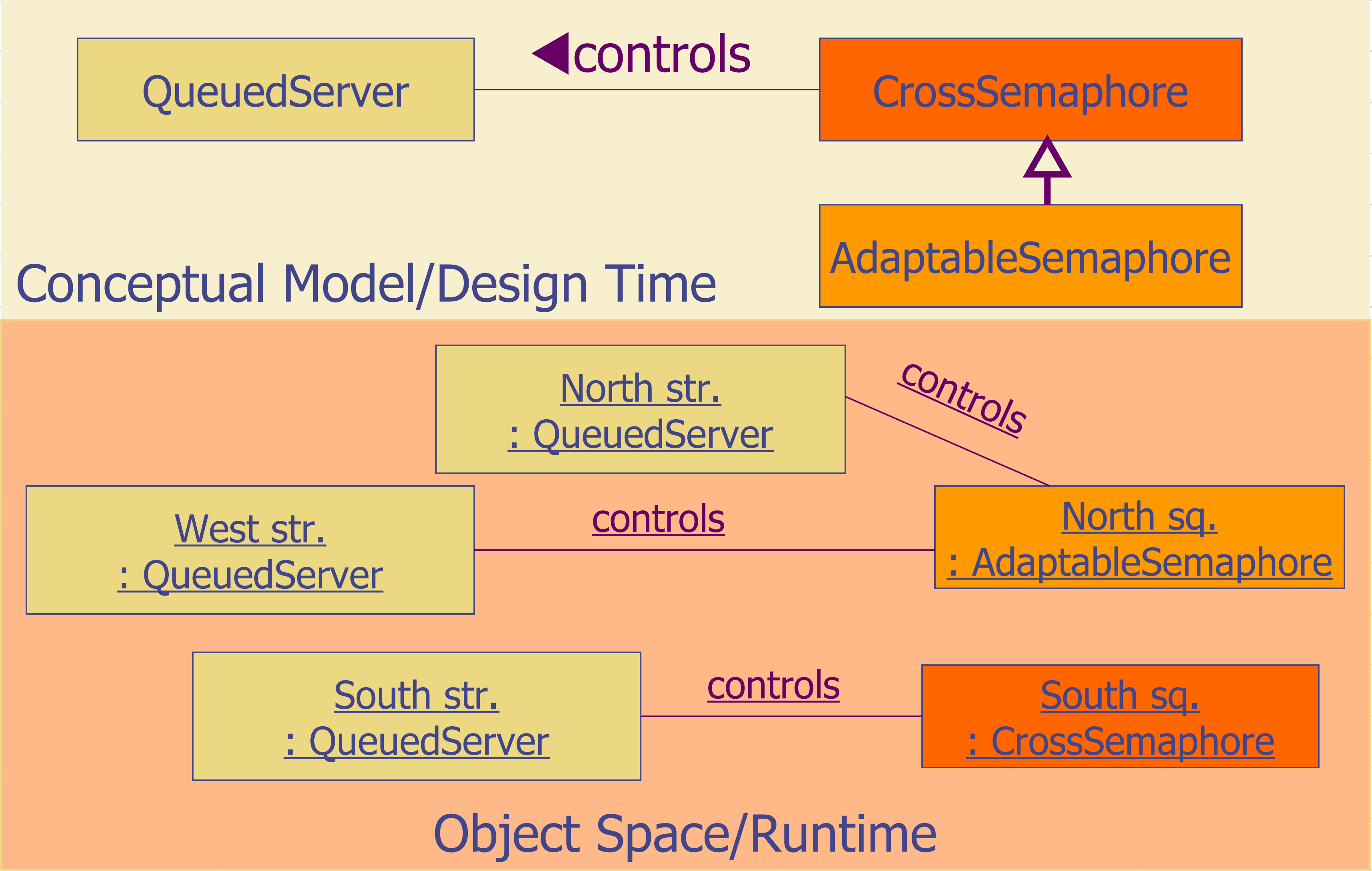
Concepts

- Generalization/specialization relationship has two significant semantic manifestations:
  - (*Inheritance*) The derived class inherits all the attributes, operations, relationships, and semantics from the base class (transitively)
  - (*Substitution*) Whenever and wherever an object of the base class is expected, an object of the derived class can occur
- If an object is a direct instance of a certain class *D*, it is said that this object is *of type* D, and also of type *B*, where *B* is a base class of *D* (transitively), and it is an (indirect) instance of *B*

Concepts

- A class can be *abstract*, meaning that it cannot have direct instances. Such a class is aimed as a generalization of other, concrete classes that can have instances. Example: ModelElement, FlowElement (names written in italic)
- A class can be derived from several base classes (multiple inheritance)
- Generalization/specialization is a conceptual relationship that exists in the conceptual model, at design time. It has the described semantic manifestation in the object space, at runtime, but does not have an explicit "instance" counterpart

Concepts


◀controls
QueuedServer
CrossSemaphore
AdaptableSemaphore
Conceptual Model/Design Time
North str.
: QueuedServer
controls
West str.
: QueuedServer
controls
North sq.
: AdaptableSemaphore
South str.
: QueuedServer
controls
South sq.
: CrossSemaphore
Object Space/Runtime

Support in UML

- UML supports generalization/specialization directly, with the semantics and notation described so far
- UML supports multiple inheritance
- UML supports the substitution rule directly and consistently, because all instances (of classes and data types) are accessed indirectly, over references (also referred to as object IDs in UML):
  - properties of objects (designated by attributes or association ends) result in (sets of) references
  - operation parameters are references
  - actions access objects through references

Support in C++

- Definition of a derived class:

```
class AdaptableSemaphore : public CrossSemaphore {
  int maxHor, maxVer;
  Time incTime;
};
```

- Support of inheritance:

```
AdaptableSemaphore* sem = new AdaptableSemaphore;
sem->tHor = 5; // Access to an inherited attribute
sem->maxHor = 30; // Access to an owned attribute
```

- Support of the substitution rule: conversion of pointers/references (upcasting), including when passing parameters

```
Derived* → Base*
AdaptableSemaphore* → CrossSemaphore*

CrossSemaphore* sem = new AdaptableSemaphore; //Conversion
sem->tHor = 5; // Access to an attribute of the base class
sem->maxHor = 30; // Incorrect!
```

Support in C++

- The substitution rule is not supported when objects are embodied by value:

```
class Model {
  ...
  CrossSemaphore sem;
  // sem is always and nothing but a direct
  // instance of CrossSemaphore,
  // and no substitution is possible!
  ...
};
```

Consequence: if substitution is needed (and it is generally in OO systems), do not use objects by values, but only by intermediaries (pointers/references)!

- C++ supports multiple inheritance, but with some subtle consequences and problems

Support in Java

- Definition of a derived class:

```
class AdaptableSemaphore extends CrossSemaphore {
  int maxHor, maxVer;
  ...
}
```

- Support of inheritance:

```
AdaptableSemaphore sem = new AdaptableSemaphore;
sem.tHor = 5; // Access to an inherited attribute
sem.maxHor = 30; // Access to an owned attribute
```

- Support of the substitution rule: conversion of references (upcasting), including when passing parameters

```
Ref to Derived → Ref to Base
Ref to AdaptableSemaphore → Ref to CrossSemaphore

CrossSemaphore sem = new AdaptableSemaphore; //Conversion
sem.tHor = 5; // Access to an attribute of the base class
sem.maxHor = 30; // Incorrect!
```

Support in Java

- The substitution rule is supported directly and consistently, without exceptions, because only references to objects and instances of built-in types can be named (objects of classes are always unnamed):

```
class Model {
  ...
  CrossSemaphore sem;
  // sem is always and nothing but a reference
  // to a (possibly indirect) instance of CrossSemaphore,
  // and substitution is always possible!
  ...
};
```

- Java does not support multiple inheritance

Advanced Concepts

- OO theory recognizes two kinds of inheritance:
  - Inheritance of interfaces, which implies substitution: the derived abstraction inherits the interface of the base abstraction, thus being capable of satisfying the same clients and substituting the generalized instances
  - Inheritance of implementations, which reduces redundancies (but does not necessarily implies substitution): the derived abstraction takes all pieces of structure and behavior from the base abstraction, but does not necessarily satisfies the same interface
- Different OO languages support different combinations of these kinds, but a commonly accepted approach nowadays (in most popular languages) is the described one with generalization/specialization, which includes both kinds
- C++ supports both kinds separately or together

Chapter 8: Operations

Motivation

Concepts

Support in UML

Support in C++

Support in Java

Advanced Concepts

Motivation

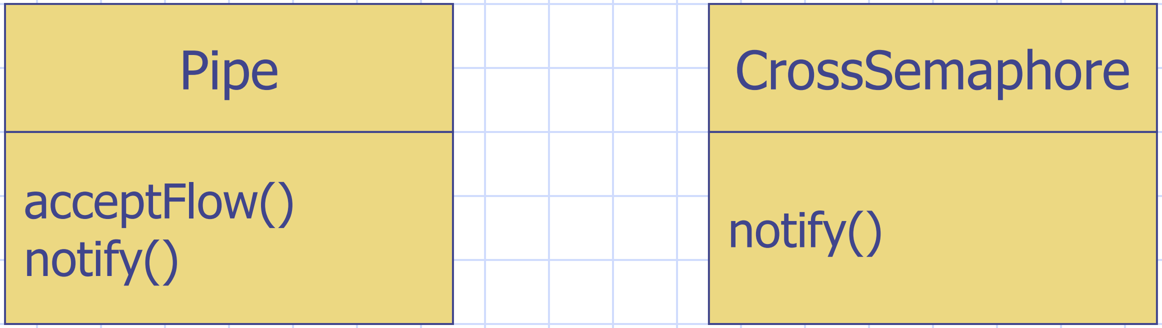
- The *structure* of TSS created so far is not sufficient to perform the main system’s task – simulation of traffic. It is only a basis for the system’s *behavior* that is built upon it.
- For example, a pipe must react on a traffic occurrence on its entrance by generating a delayed occurrence on its exit.
- A cross semaphore must change the open direction on the notification of the passage of a time interval.
- Similarly, other objects in the system must provide various services to other objects.

Concepts

- *Operation* is the specification of a service that can be requested from any object of the class in order to affect behavior
- *Method* is an implementation of an operation
- An operation is an element of the conceptual model, i.e., a member of a class
- An operation specifies that a service may be requested from any direct or indirect instance of that class

Concepts

- An operation has its name, may have its formal arguments (parameters), and possibly its return type
- At runtime, an operation of an object may be invoked. The actual arguments are then supplied
- The invocation of the operation is manifested by the behavior specified by the corresponding method of the class the object belongs to, no way how the object was accessed (i.e., possibly as an instance of a base class)



Pipe

acceptFlow()
notify()

CrossSemaphore

notify()

Support in UML

- UML supports operations and methods as described, with the following notation of operation specification:

  ```
  operationName (argumentList) : returnType
  ```

  where **`argumentList`** and **`returnType`** are optional, and **`argumentList`** is a comma-separated list of argument specifications:

  ```
  argName : argType = defaultValue
  ```

- Actual arguments are always references to objects (of classes or data types)
- Methods are defined in terms of *actions*

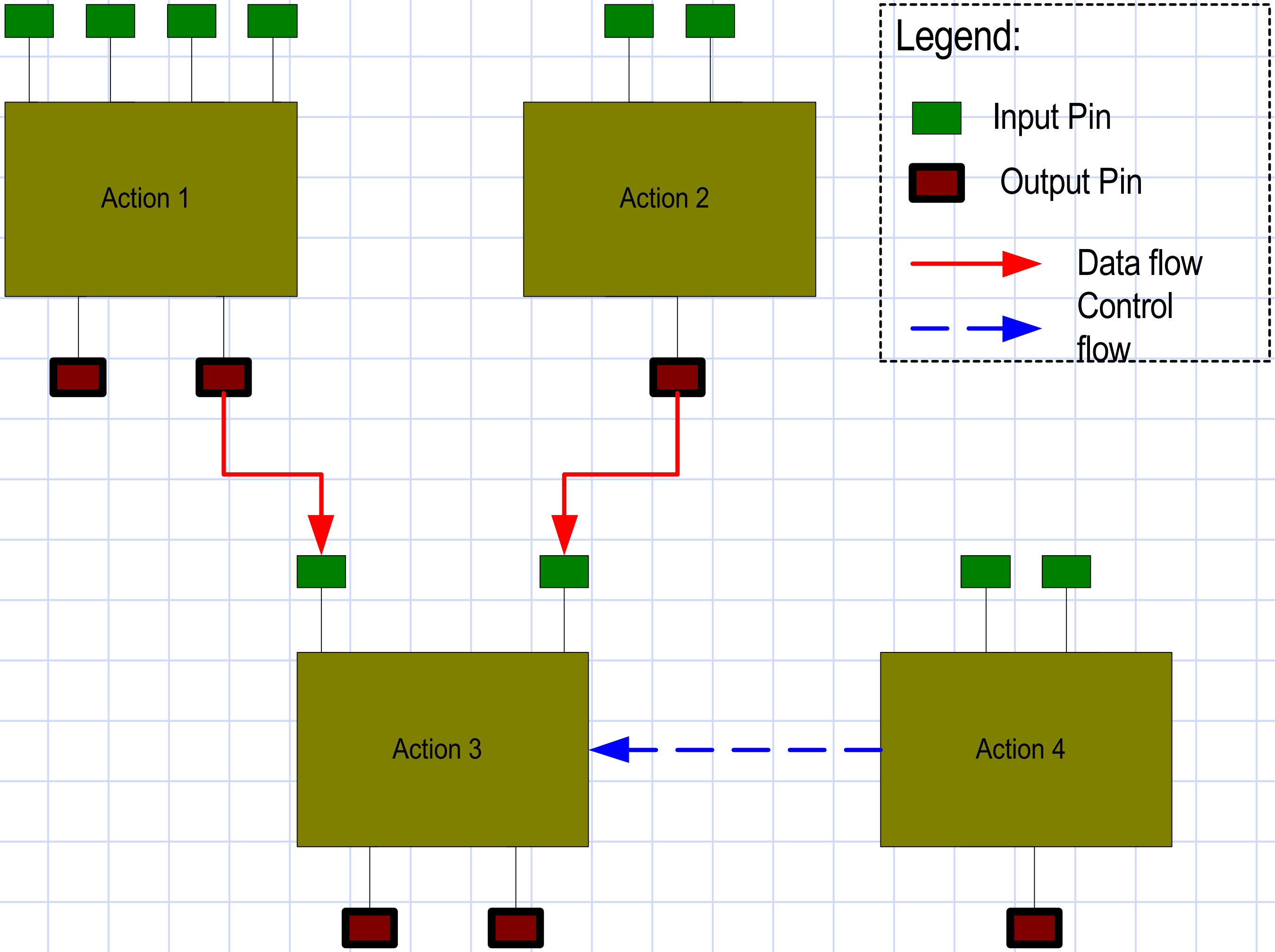
Support in UML

- *Action* is a unit of behavior used to construct methods
- An action can read and/or modify a part of the object space in a consistent manner
- A method is a complex mesh of actions, connected by control and object flow to provide a complex execution of the actions

Support in UML

- A read/write action is an atomic action that represents a request to perform an atomic access to or modification of the object space
- There are read/write actions of different kinds, such as "Create object," "Delete object," "Modify attribute value," "Create link," “Read links,” or "Delete link"
- Each action can have its *pins*. Pins represent the parameters of an action
- UML action semantics uses the combined data-flow and control-flow paradigms of action execution

Support in UML



Support in UML

- UML action semantics defines actions for invoking an operation of an object over a reference:
  - with synchronous invocation (the caller action does not complete until the invoked method is completed)
  - with asynchronous invocation (the caller action completes without waiting for the invoked method to complete)

Support in C++

- Operation specification:

```
class Pipe {
  ...
  void acceptFlow(); // Accepts a vehicle occurrence
  void notify(); // Notifies that the delay has elapsed
  ...
};
```

- Method definition uses the traditional procedural programming style (as in the C language):

```
void Pipe::acceptFlow () {
  // Access to an (inherited) attribute
  // of the object of which the method is invoked:
  counter = counter+1;
  // Request notification after the delay
  // by calling an operation of the same object:
  raiseEvent();
}
```

Support in C++

- Method invocation:

```
void Pipe::notify () {
  // If there is a target, send the vehicle to it:
  if (target!=0) target->acceptFlow();
}
```

- Method bodies may contain expressions and statements in a usual procedural style (sequences, conditions, loops, etc.)
- C and C++ recognize functions only; procedures are a special case of functions with no return value (**`void`** as the return type)
- Arguments can be passed by value or by reference
- Only synchronous call is supported
- C++ allows ordinary non-member (global) subprograms (vertical compatibility with C)

Support in Java

- Very similar to C++ (except for some slight notational differences)
- The language for programming methods is almost equal to C/C++
- Arguments of built-in types are passed by value, and objects of classes by reference (substitution is implied)
- Calls are always synchronous
- No non-member (global) functions are allowed

Advanced Concepts

- Problem: how to ensure that an object has a proper initial state when it is created, e.g., its attribute values are properly initialized?
- Possible solution:

```
class Pipe {
  ...
  void init();
  ...
};

void Pipe::init() {
  counter = 0;
  ... // Other necessary initializations
}

// When an object is created,
// init() must be called immediately:
Pipe* aPipe = new Pipe;
aPipe->init();
```

Advanced Concepts

- This approach is error-prone, because programmers can easily miss to invoke the **`init()`** operation!
- C++ and Java support *constructors*. A constructor is a member function (named the same as its class) that is implicitly invoked every time an object is created. The compiler ensures constructor invocation at all places of object creation, so that no errors can occur:

```
class Pipe {
  Pipe();  // Constructor
  ...
};

Pipe::Pipe() {
  counter = 0;
  ... // Other necessary initializations
}

Pipe* aPipe = new Pipe; //Constructor is called implicitly
```

Advanced Concepts

- Constructors are alike other member operations in many aspects:
  - they have **`this/self`** reference (to be explained soon)
  - they may have arguments as any other operation (to provide parameterized construction)
  - a class may have several constructors, provided they differ in number and types of arguments
- When an object is constructed, the constructor of the base class is invoked before the constructor of the derived class is executed (and so on transitively)
- C++ and Java support *destructors* also: these are operations that are implicitly called on object destruction

Advanced Concepts

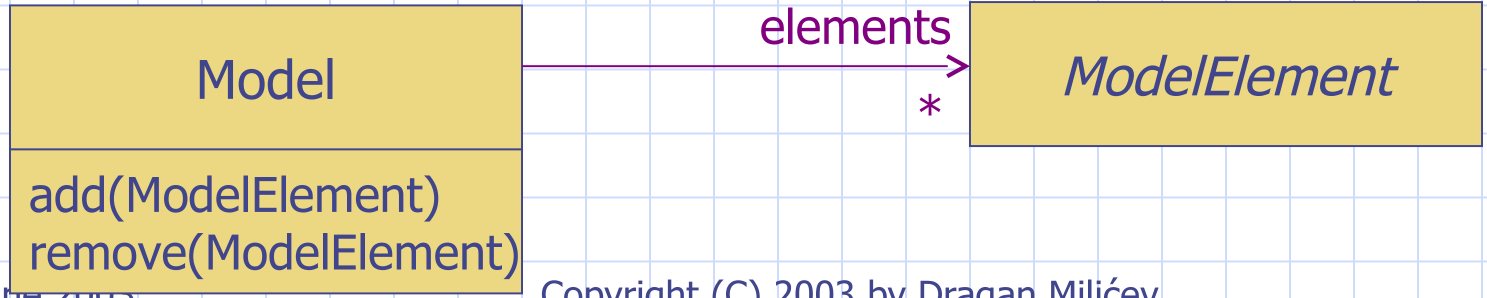
- In the scope of a method of a class, there is an implicitly defined reference to the object for which the method is invoked:
  - called **`self`** in UML
  - called **`this`** in C++ (pointer to the object) and Java
- Every direct access to a member of the object (property or operation) is actually an indirect access over this reference:

```
void Pipe::acceptFlow () {
  counter = counter+1; // implicitly this->counter
  raiseEvent(); // implicitly this->raiseEvent()
}
```

Advanced Concepts

- The **`self`/`this`** reference may be used to pass the reference to the server object to other objects, in order to create links to it:

```
Pipe::Pipe () {
  // Sign this newly created pipe
  // to the entire model;
  // Model::add() accepts an argument
  // of type ModelElement*;
  // theModel is a globally accessible reference
  // to an object of Model:
  theModel->add(this);
}
```



Advanced Concepts

- An operation can be of *class scope* (instead of *instance scope*), meaning that it is a service of the class, not of a particular object, and it can be called without specifying the server object
- UML, C++, and Java support operations of class scope directly (called *static* operations in C++ and Java)
- An operation can be *query*, meaning that it does not modify the state of the server object
- UML and C++ support query operations (called constant member functions in C++)

Chapter 9: Polymorphism

Motivation

Concepts

Support in UML

Support in C++

Support in Java

Motivation

In TSS, any *flow element* (e.g. a pipe or a flow source), can have any other flow element (e.g. a switch or a flow sink) as its *flow target*. When the former generates a vehicle occurrence at its output, the target flow element must *accept* that flow occurrence through its corresponding service (operation).

However, different kinds of flow elements process the accepted flow in completely different ways. For example, a flow sink simply counts the occurrence and does not propagate it, while a pipe propagates it after a certain delay.

How to provide different behavior of different kinds of flow elements for the same service "accept a flow occurrence," so that the client (source) elements do not depend on the type of the server (target) elements and that the targets can be attached to sources interchangeably?

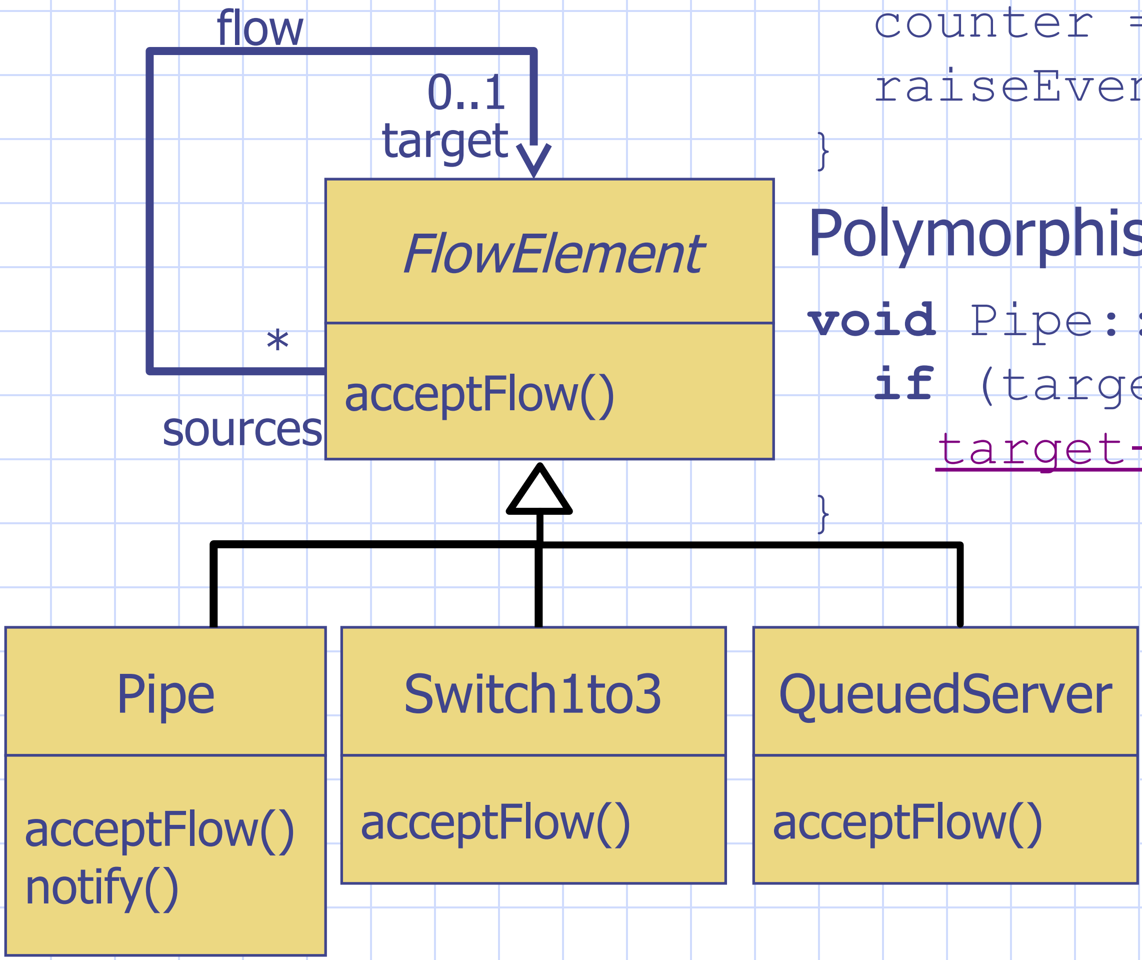
Concepts

- A derived class may *redefine* (or *override*) an operation of its base class
- This means that the derived class provides another implementation (i.e., method) for the same operation, offering different behavior for the same service
- If a class does not override an operation, it inherits the method of that operation from its base class (transitively)
- An operation may be *abstract*, meaning that its implementation is not provided in the class. Such a class is then also abstract

Concepts

- At runtime, an operation of an object may be invoked. The invocation of the operation is manifested by the behavior specified in the corresponding method of the class to which the object belongs, regardless to how the object was accessed (i.e., possibly as a kind of an object of a base class)
- In other words, a client that invokes an operation of a server object can access the server object as a generalized entity (i.e., as an object of the base class). In that case, the method of the derived class will be invoked. This mechanism is called *polymorphism*

Concepts

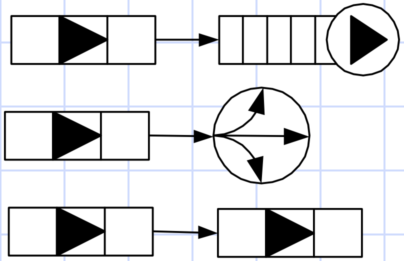


Redefined operation:

```
void Pipe::acceptFlow () {
   counter = counter+1;
   raiseEvent();
}
```

Polymorphism at invocation place:

```
void Pipe::notify() {
   if (target!=0)
     target->acceptFlow();
}
```



Concepts

- The purpose of polymorphism is to make the clients that invoke an operation independent of the variation of the operation's implementation
- The client is spared from knowing the specialties about the server–the client tends to regard the server as a generalized thing and to access it through its generalized interface
- The specialties of different kinds of servers are incorporated in the polymorphic operations and their overridden derivatives
- This way, the interfaces between clients and servers become looser, and therefore more controllable
- This is a key point to constructing flexible software

Concepts

- This is because a modification of the behavior of the client side can be achieved by *adding* parts of software (i.e., overriding operations in derived classes), and not *modifying* parts of software, which is always error-prone and risky
- The client does not experience any modification if a new class at the server side is added in the hierarchy or a polymorphic operation is overridden in a derived class in the server-side hierarchy, and yet the software behaves differently
- This mechanism is one of the most important contributions of the object-oriented programming paradigm

Support in UML

- UML directly supports polymorphism, but does not specify the rule for method resolution, because it is done differently in different implementation languages
- To override an operation, the same operation (with the same name, arguments, and return type) is specified in the derived class, with its own method, and with the "redefine" relationship to the overridden operation
- Operations are polymorphic by default. If an operation should not be polymorphic, it is tagged as **`leaf`**. All operation invocations are polymorphic

Support in C++

- In C++, operations are non-polymorphic by default. To be polymorphic, an operation must be specified as *virtual* at least in the base class:

```
class ModelElement {
  ...
  virtual void acceptFlow();
};
```

- An operation call is polymorphic if the server object is accessed over a pointer/reference:

```
 void Pipe::notify() {
  if (target!=0) target->acceptFlow();
}
```

- Abstract operations are specified with **=0**:

```
class ModelElement {
  ...
  virtual void acceptFlow() = 0;
};
```

Support in Java

- In Java, operations are polymorphic by default. To be non-polymorphic, an operation must be specified as *final* :

```
class ModelElement {
   ...
   final void acceptFlow() {...}
}
```

- Operation calls are always polymorphic, since a server object is always accessed over a reference:

```
if (target!=null) target.acceptFlow();
```

- Abstract operations are specified with **`abstract`**:

```
class ModelElement {
   ...
   abstract void acceptFlow();
}
```

Chapter 10: Encapsulation

Motivation

Concepts

Support in UML

Support in C++

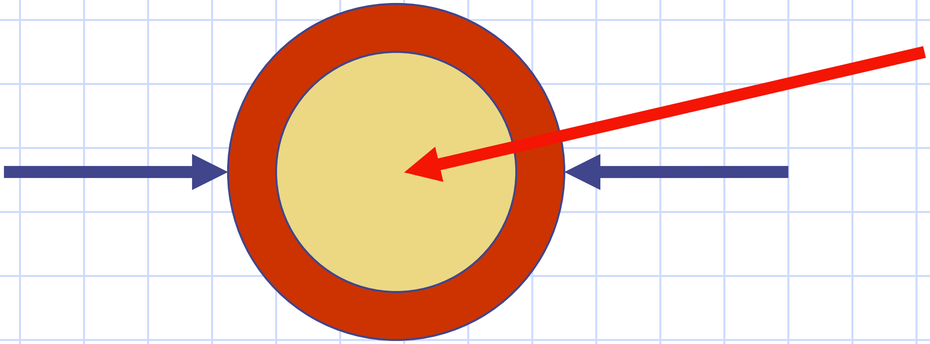
Support in Java

Motivation

- It has been assumed so far that all members of classes are freely accessible from anywhere in the program. However, this can be error prone, because there is no protection from accidental or intentional corruption of object states. For example, a client can modify the counter of transported vehicles of a pipe incorrectly.
- Besides, such software is very likely to be inflexible. If a client relies on a part of the server's implementation, the server's implementation cannot be modified without affecting the client (the domino effect is very likely).
- These issues are caused by the fact that the interactions between objects are not specified in a controlled manner, so that violations of defined interactions cannot occur.

Concepts

- One of the fundamental principles of software engineering adopted by OO technology through first-class concepts is *encapsulation*
- Encapsulation encompasses the following assumptions:
  - For a software component (class, module, package, or whatever piece of software for which encapsulation is available), two parts can be distinguished: *interface* and *implementation*
  - Other software components can access only the interface of the component; its implementation is encapsulated, hidden, and inaccessible to the clients

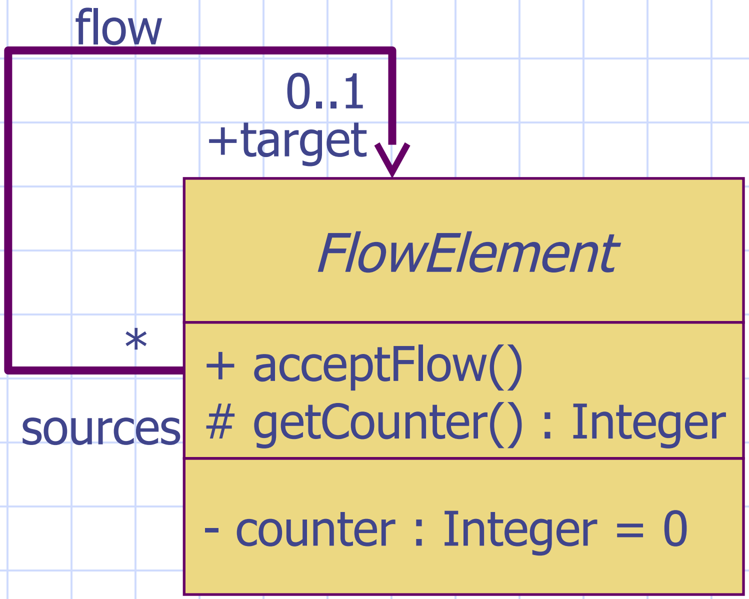


Concepts

- Different languages support encapsulation in different ways and at different levels of granularity
- Most of them support encapsulation at class level, whereby each member of a class can be:
  - public: available from anywhere (constitutes the public interface of the class)
  - protected: available from the scope of the same class and derived classes only (constitutes a restricted interface to derived classes as “privileged” clients)
  - private: available only from the scope of the same class (constitutes the implementation of the class)

Support in UML

- UML supports the described levels of visibility (accessibility) of class members, including properties (attributes and association ends) and operations. Notation:
  - public: +
  - protected: #
  - private: -



Support in UML

- *Package* is a general grouping mechanism in UML. A package owns model elements, such as classes, associations, and other packages, too. This way, packages are used to organize the model hierarchically
- UML supports encapsulation of package elements, too, whereby an element of a package can be (the same notation as for class members):
  - public: accessible from anywhere
  - private: accessible from the same package only

Support in C++

- C++ supports accessibility levels of class members as described (public, private, protected):

```
class FlowElement : public ModelElement {
public:
  virtual void acceptFlow();
protected:
  FlowElement();
  // A protected constructor implies that the class
  // is abstract, because direct instances of it
  // cannot be created!
  int getCounter(); // Returns the value of counter
private:
  int counter;
};
```

- From outside the class, it is no more possible:

```
Pipe* aPipe = new Pipe;
aPipe->counter = 5; // Compilation error!
```

Support in Java

- Java supports accessibility levels of class members as described (public, private, protected). Each class member must have the accessibility specifier in front of its declaration
- Java supports packages and accessibility of their elements (classes and nested packages):

```
public abstract class FlowElement
  extends ModelElement
{
    public void acceptFlow() {...}
    protected FlowElement() {...}
    public int getCounter() { return counter; }
    private int counter = 0;
}
```

Chapter 11: Interfaces

Motivation

Concepts

Support in UML

Support in C

Support in Java

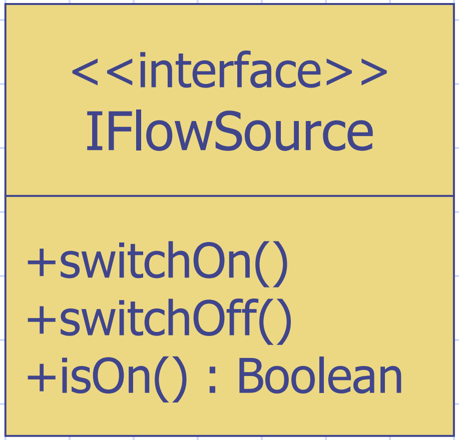
Advanced Concepts

Motivation

- In TSS, we can generally allow other kinds of flow elements (but not all) to be controlled by semaphores. For example, besides queued servers, why shouldn't we allow semaphores to control flow sources, too. In general, semaphores only require that the controlled elements provide the "switch on" and "switch off" services.
- Similarly, a scheduler of timed events must notify different kinds of model elements at proper moments (e.g., delay of vehicles in pipes, change of direction in semaphores, etc.). All that the scheduler needs is that the servers respond to the "notification" message.
- In general, we need a concept that allows loose coupling of classes, whereby the client class relies only on the assumption that the server provides a set of services.

Concepts

- *Interface* is a collection of operations that are used to specify a service of a class or component



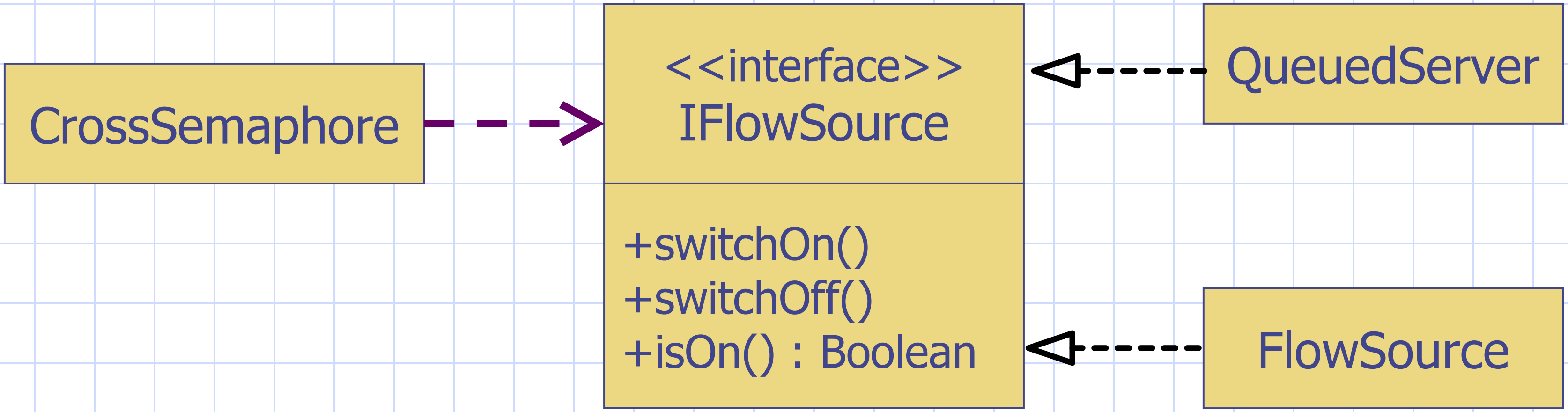
<<interface>>
IFlowSource

+switchOn()
+switchOff()
+isOn() : Boolean

- Interface is a specification of *obligations* that the server side fulfills, and the client side requires for a successful collaboration. Therefore, an interface defines a *contract* between two interested parties
- Interface consists of abstract operations only and has no structure and no methods

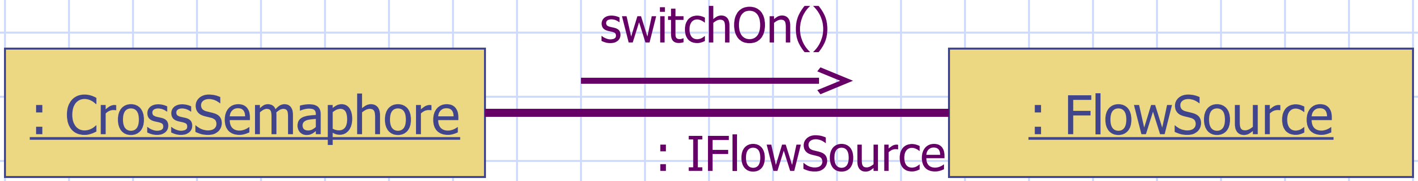
Concepts

- A class or a component may *realize* a set of interfaces, offering the specified services to the clients, meaning that it implements the operations of the interfaces providing the methods for them
- A class or a component may *depend* on a set of interfaces, meaning that it requires from its clients to realize those interfaces (provide methods for the operations)



Concepts

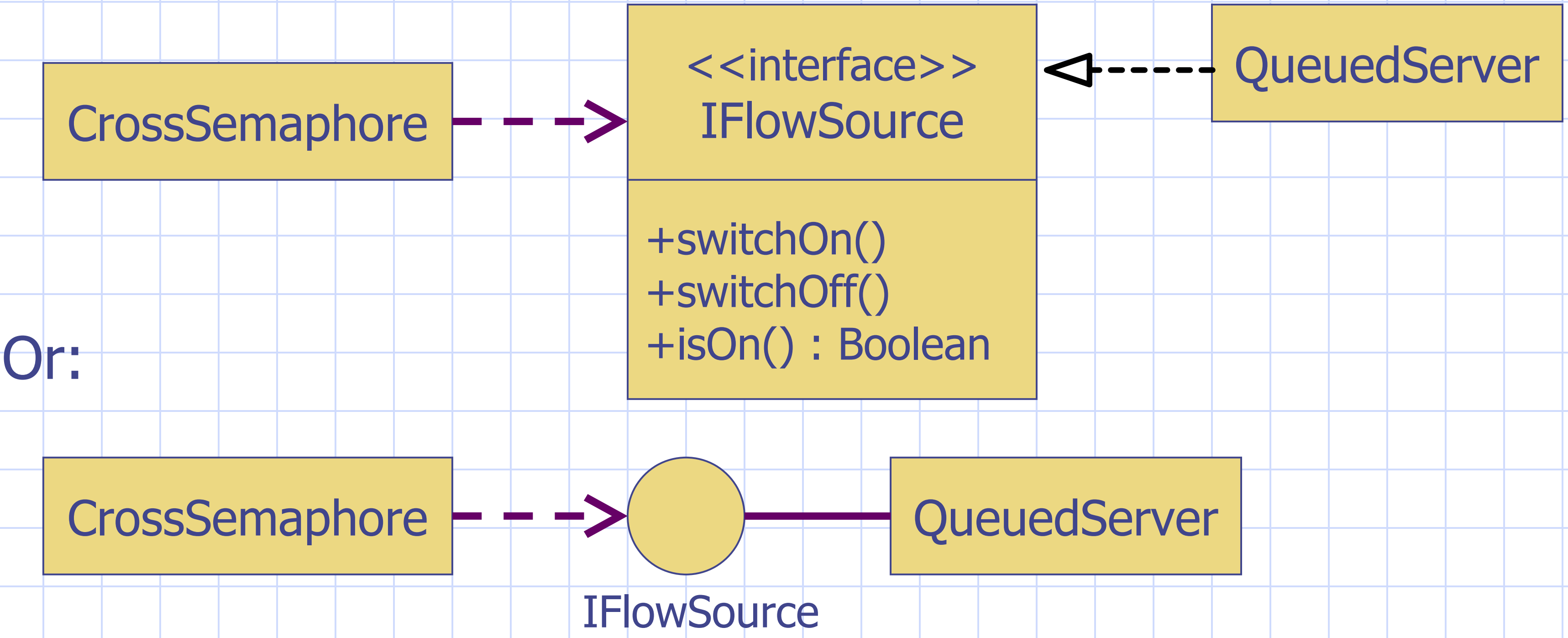
- Interface realization (implementation) implies substitutability: any object whose class realizes the required interface can be the server of an object that depends on the interface



- Interface consists of abstract operations only, thus allowing extremely loose coupling between software components
- Interfaces can be specialized, meaning that the specialized interface inherits and extends the generalized interface (interface inheritance)

Support in UML

- UML fully supports interfaces. All classifiers may realize interfaces (class, data type, component, node)
- Alternative notations:



Support in C++

- C++ does not support interfaces directly
- An interface can be specified by an abstract class with nothing but abstract operations:

```
class IFlowSource {
public:
  virtual void switchOn() = 0;
  virtual void switchOff() = 0;
  virtual void isOn() = 0;
};
```

- Implementation of interfaces is done by derivation of classes:

```
class QueuedServer : public FlowElement,
                     public IFlowSource {
public:
  virtual void switchOn();
  virtual void switchOff();
  virtual void isOn();
  ...
};
```

Support in Java

- Java supports interfaces directly and completely. Although a class may extend (specialize) only one base class, it can implement (realize) many interfaces
- Specification of an interface:

```
interface IFlowSource {
  void switchOn();
  void switchOff();
  void isOn();
}
```

- Implementation of interfaces:

```
public class QueuedServer extends FlowElement
                          implements IFlowSource {
  public void switchOn() {...}
  public void switchOff() {...}
  public void isOn() {...}
  ...
}
```

Advanced Concepts

- Similarities between interfaces and (possibly abstract) classes:
  - both are classifiers, because they designate sets of instances
  - both have operations as specifications of services that may be requested from the instances
  - both can be specialized and generalized
- Differences between interfaces and classes:
  - unless it is abstract, a class may have direct instances; instances of interfaces are always indirect
  - even when it is abstract, a class may have properties (attributes and association ends) and methods; interfaces consist of abstract operations only
  - a class specifies a contract in a concrete way, with (most often) some implementation of the contract; interfaces specify contracts in a pure, abstract way

Chapter 12: Interactions

Motivation

Concepts

Support in UML

Support in C++ and Java

Motivation

The behavior of TSS is provided by interactions between objects, whereby objects invoke operations in a complex manner. Some of these interactions lay in the core of the system’s behavior, making its key mechanisms.

One of the key mechanisms in TSS is the simulation of discrete events in time. Some model elements, e.g. flow sources, pipes, queued servers, and semaphores, generate timed *events*, which carry the information about the time at which they should be notified to react in some way. A centralized *scheduler* holds a list of raised events, sorted chronologically, simulates the passage of time, and notifies the corresponding model elements by handling the events in the chronological order.

Motivation

- For example, when a flow source is notified of an event, it generates a vehicle occurrence, transports it to its target, and then raises another event scheduled at a future moment after a random delay relative to the previous event, in order to be notified to generate a new vehicle occurrence and repeat the same procedure.
- Similarly, when a pipe accepts a flow occurrence, it raises an event with a random delay. When it is notified about the passing of the delay, it generates a vehicle occurrence to its target, thus simulating the transportation delay of vehicles.
- Generally, it would be useful to have a means to visualize, specify, construct, and document the scenarios of interactions between objects in the software system.

Concepts

- *Interaction* is a behavior that comprises a set of messages that are exchanged among a set of objects within a particular context to accomplish a purpose
- Objects interact according to the designed *scenarios*, which are specific sequences of actions that illustrate behavior
- Interactions are used to model dynamic aspects of collaborations, representing societies of objects playing specific roles, all working together to carry out some behavior that is bigger than the sum of the elements
- Those roles represent prototypical instances of classes, interfaces, or other classifiers, and their dynamic aspects are visualized, specified, constructed, and documented as flows of controls that may encompass simple, sequential threads through a system, as well as more complex flows that involve branching, looping, recursion, and concurrency

Concepts

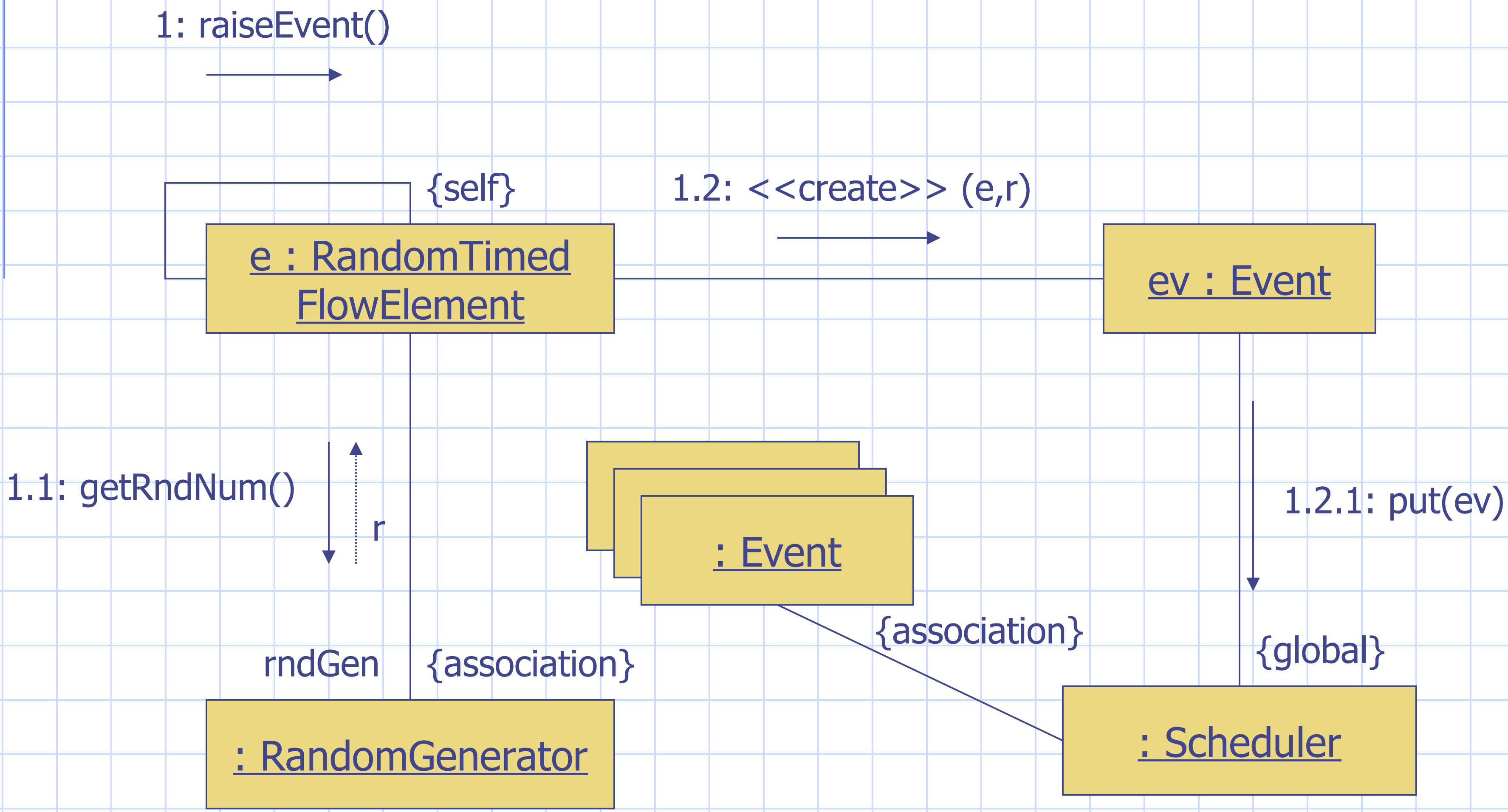
- An interaction has its *context*:
  - a collaboration of objects in a part of the system (e.g., a key mechanism of the system)
  - a method, where objects local to the method and globally accessible to the method collaborate to provide the implementation of an operation
  - behavior of a class, where attributes of a class and other globally accessible objects collaborate to provide the behavior of the class
- Interactions consist of:
  - objects and roles that take part in the interaction
  - links between objects
  - messages (e.g. operation calls) that flow over links between objects

Support in UML

- UML supports interactions in a very complex and flexible way. Interactions are depicted in *interaction diagrams*
- There are two kinds of interaction diagrams, which depict the same interaction from two different viewpoints:
  - *collaboration diagram* has a shape of a graph of objects and links with flow of messages over the links, thus emphasizing the structural connections between participants
  - *sequence diagram* has a shape of a timeline, where objects are placed across the *x* axis and messages are placed along the *y* axis, thus emphasizing the time ordering and focus of control
- Collaboration and sequence diagrams are semantically equivalent, because they rely on the same information and depict the same interaction, emphasizing its different details

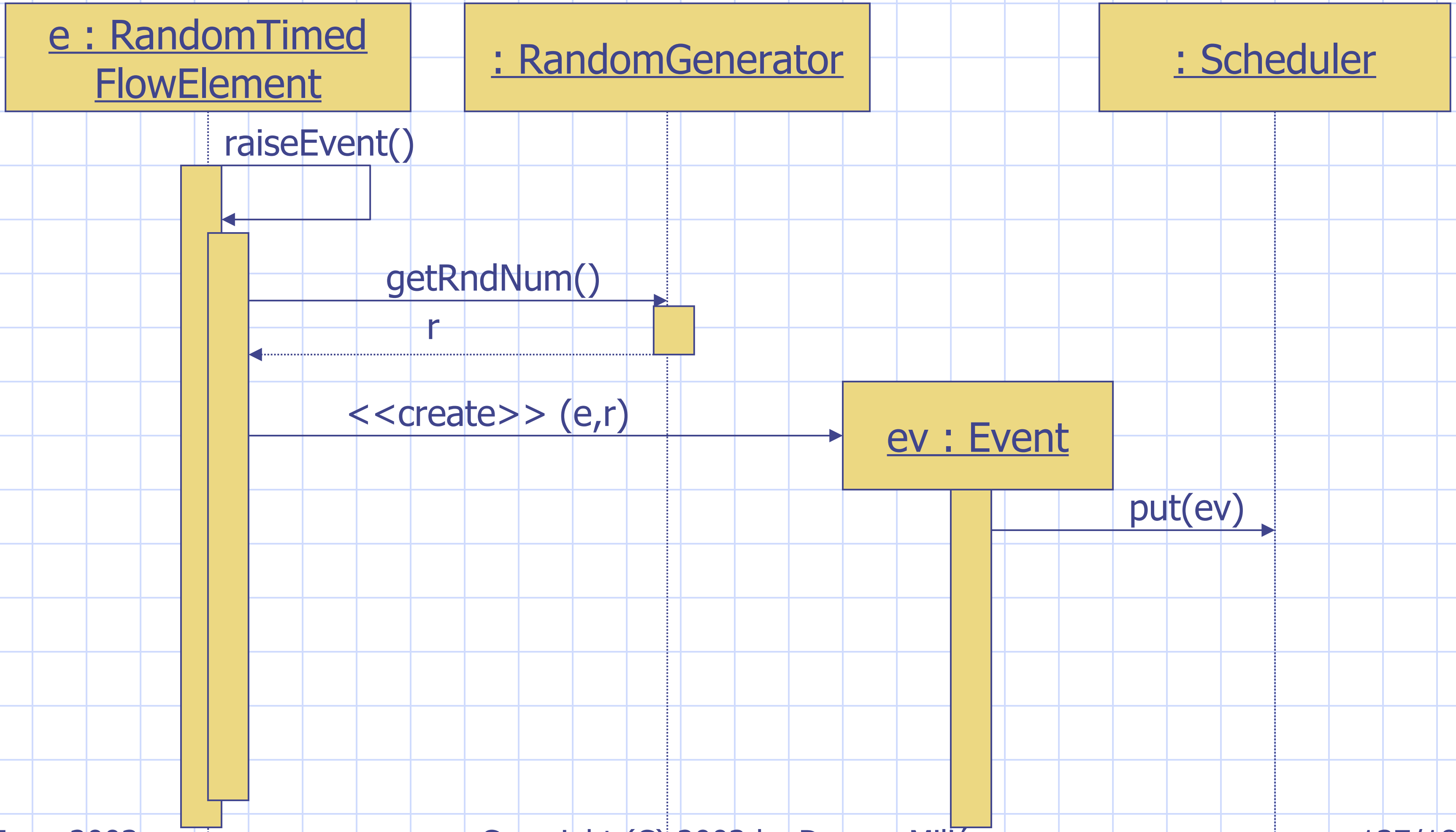
Support in UML

Collaboration diagram for raising an event



Support in UML

Sequence diagram for raising an event



Support in UML

- UML interaction diagrams allow specifications of many other details:
  - creation and deletion of objects and links
  - different flow of control issues (branching, looping, recursion)
  - threads of control
- Although very rich in concepts, UML interaction diagrams do not have fully formal semantics, so they cannot be executable in a general case
- A modeling tool may help in generating implementation code from interaction diagrams in some special cases

Support in C++ and Java

- C++ and Java do not support specification of interactions directly, but interactions are spread across the implementations of many operations:

```
void RandomTimedFlowElement::raiseEvent () {
  Time tm = 0;
  if (rndGen) tm = rndGen->getRandom(); else return;
  new Event(this,tm);
}

Event::Event (ITimedElement* targetElement, Time tm)
  : time(tm), target(targetElement) {
  theScheduler->put(this);
}
```

- This is one of the most important disadvantages and restrictions of OO programming in languages like C++ and Java – poor readability of scenarios and key mechanisms

Part III: Design Patterns

About Design Patterns

Singleton

Strategy

Template Method

Visitor

Composite

Chapter 13: About Design Patterns

What are Design Patterns?

Pattern Description

Pattern Classification

What are Design Patterns?

- *Design patterns* are simple and elegant solutions to specific, commonly recognized problems in OO software design
- DPs are not concrete, physical *artifacts* of software development. Instead, they are *ideas* and *directions* to solve a design problem
- DPs are expressed in terms of *collaborations*, which consist of the *structure*, expressed in terms of abstract roles that take part in collaborations to accomplish a purpose, and the *interactions* of those participants
- In every concrete problem, the developer must concretize the roles to specific participants in the system

What are Design Patterns?

- DPs cannot be taken as ready-to-use pieces of code, model, or any other artifact. They are reused as solutions and ideas instead to help developers to make clear and stable OO design
- DPs are crucial for software reuse
- However, DPs are not like other means of reusability:
  - algorithms: known procedures to solve some problems
  - libraries: units of code ready to be used as parts of the system, consisting of subprograms, types, etc.
  - frameworks: ready-to-use collaborations of classes and objects that provide certain mechanisms in a specific problem domain

Pattern Description

- A design pattern has four essential elements:
  - Name: an identifier of a pattern; a handle we can use to describe a design problem, its solution, and its consequences; used to increase the understandability among developers
  - Problem: describes when to apply the pattern, i.e., a commonly recognized problem and its context
  - Solution: describes the elements that make up the design, i.e., a collaboration, consisting of the structure (roles, responsibilities, relationships) and behavior (interactions)
  - Consequences: the results and trade-offs of applying the pattern

Pattern Classification

- According to their purpose, DPs can be:
  - creational: concern the process of object creation
  - structural: deal with the composition of classes and objects
  - behavioral: characterize the ways in which classes or objects interact and distribute responsibility
- According to their scope, DPs can be:
  - class: deal primarily with classes and their relationships
  - object: deal with dynamic connections and interactions between objects
- The rest of the Tutorial presents a small set of selected DPs from the referential book of "the gang of four" (Gamma et al.), explained using the TSS example

Chapter 14: Singleton

Motivation

Intent and Applicability

Structure and Collaborations

Implementation

Consequences

Motivation

In TSS, the class `Scheduler` should have exactly one instance, because we want to have a centralized, globally accessible object of that class to accept and handle events.

This requirement can be refined as follows:

- the class has one instance that is properly initialized and certain to exist when it is accessed
- there is no way to create more instances accidentally or on purpose
- the sole instance is easily and globally accessible.

A global reference variable makes an object accessible, but it does not keep from creating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created.

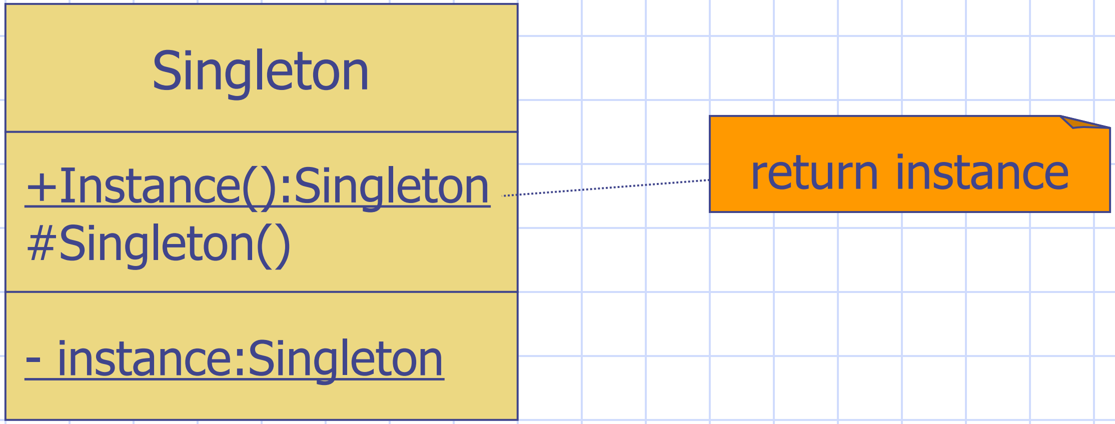
Intent and Applicability

- Intent (object/creational) :

  Ensure a class has exactly one instance, and provide a global point of access to it

- Applicability: use the Singleton DP when
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Structure and Collaborations



- Participants:
  - **Singleton** (Scheduler):
    - defines an Instance operation that lets clients access its unique instance; Instance is a class-scope operation (static)
    - may be responsible for creating its own instance
    - is responsible for keeping from creating more instances
- Collaborations:
  - Clients access a Singleton instance solely thought Singleton's Instance operation

Implementation

- Class definition (Java):

```
class Scheduler {
  // The sole access point:
  public static Scheduler Instance() {
    return instance;
  }
  //... Other operations are possible:
  public void put(Event) {...}
  //...
  // Protected constructor prevents from
  // creating more objects, but allows subclassing
  protected Scheduler() {...}
  // The sole instance of the class:
  private static instance = new Scheduler;
}
```

- Access to the singleton object (Java):

```
Scheduler.Instance().put(this);
```

Consequences

- Controlled access to sole instance: because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it
- Reduced name space: it is an improvement over global variables that pollute the global namespace
- Permits refinement of operations: the Singleton class may be subclassed, and it is easy to configure an application with an instance of a derived class, even at runtime
- Permits variable number of instances: it is easy to have more than one, but a controlled number of instances
- More flexible than class-scope operations that can be also used to package a singleton’s functionality; however, class-scope operations cannot be polymorphic and it is not possible to have more than one instance

Chapter 15: Strategy

Motivation

Intent and Applicability

Structure and Collaborations

Consequences

Motivation

In TSS, many flow elements generate events at random time intervals. They all need generation of random numbers.

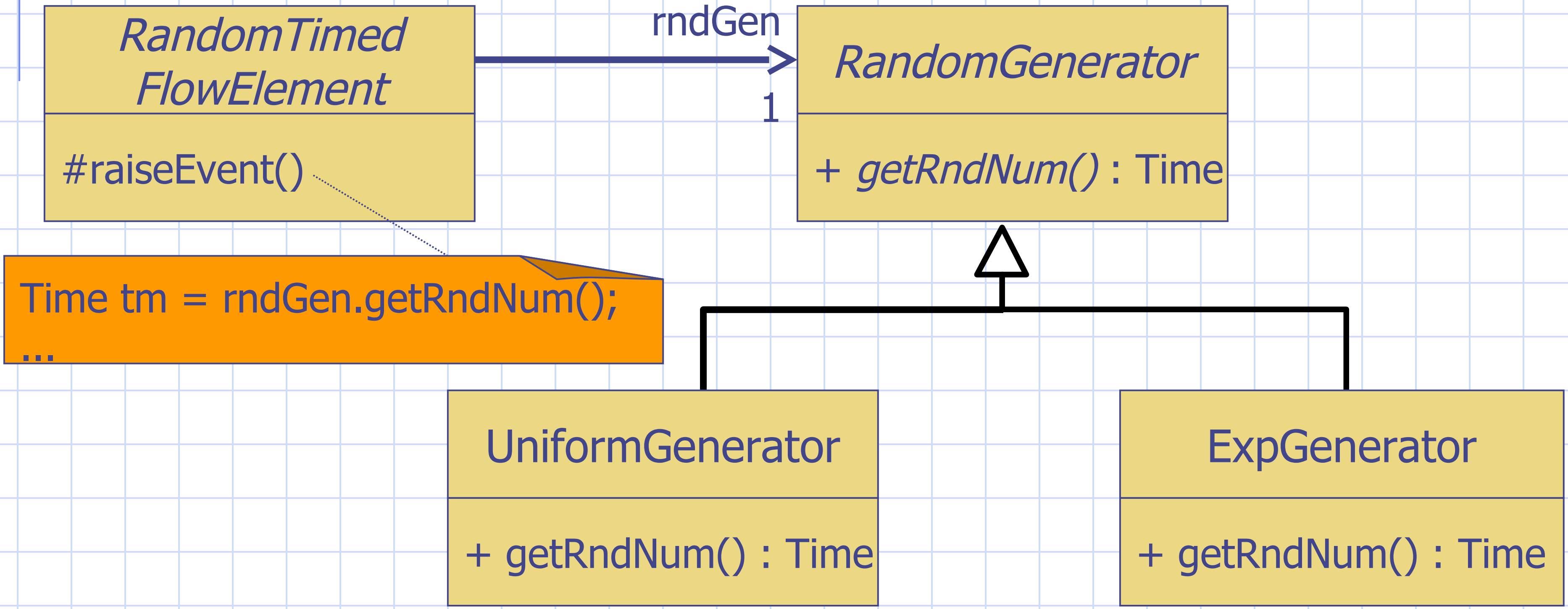
However, in a general case, they may need random numbers of different distribution (uniform, exponential, etc.), which are generated using different algorithms.

Hard-wiring all such algorithms into the classes that require them is not desirable for several reasons:

- clients that need random numbers get more complex if they include the random number generation code; they become bigger and harder to maintain
- different algorithms will be appropriate at different times; we don't want to support multiple algorithms if we don't use them at all
- it's difficult to add new algorithms and vary existing ones when random number generation is part of a client.

Motivation

- These problems can be avoided if we define classes that encapsulate different random number generation algorithms. An algorithm that is encapsulated in this way is called a *strategy*.



RandomTimed FlowElement

#raiseEvent()

```
Time tm = rndGen.getRndNum();
...
```

rndGen

1

RandomGenerator

+ *getRndNum()* : Time

UniformGenerator

+ getRndNum() : Time

ExpGenerator

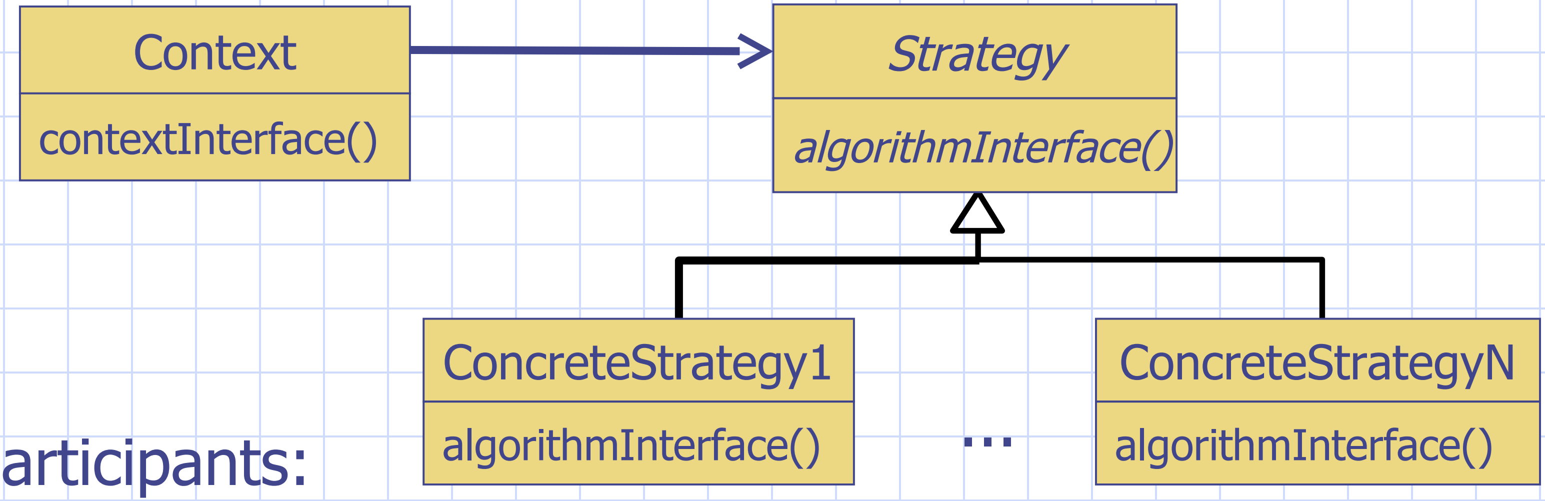
+ getRndNum() : Time

Intent and Applicability

- Intent (object/behavioral):

  Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Also known as *Policy*
- Applicability: use the Strategy DP when:
  - many related classes differ only in their behavior; Strategies provide a way to configure a class with one of many behaviors
  - you need different variants of an algorithm, e.g. reflecting different time/space trade-offs
  - an algorithm uses data that clients should not know about; use the Strategy DP to avoiding complex, algorithm-specific data structures
  - a class defines many behaviors, and these appear as multiple conditional statements in its operations; instead of many conditionals, move related branches into their own Strategy class

Structure and Collaborations



- Participants:
  - **Strategy** (RandomGenerator)
    - declares an interface common to all supported algorithms; Context uses this interface to call the algorithm defined by ConcreteStrategy
  - **ConcreteStrategy** (UniformGenerator, ExpGenerator)
    - implements the algorithm using the Strategy interface
  - **Context** (RandomTimedFlowElement)
    - is configured with a ConcreteStrategy object
    - maintains a reference to a Strategy object
    - may define an interface that lets Strategy access its data

Structure and Collaborations

- Collaborations:
  - Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass a reference to itself as an argument to Strategy operations. That lets the strategy call back on the context as required
  - A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from

Consequences

- Benefits:
  - Reusable families of related algorithms, factored by inheritance
  - An alternative to subclassing the Context class directly to give it different behavior, by hard-wiring the behavior into Context, thus making Context harder to understand, maintain, and extend
  - Strategies eliminate conditional statements
  - A choice of implementation among different implementations of the same behavior
- Drawbacks:
  - Communication overhead between Strategy and Context
  - Increased number of objects, especially if strategies are non-shareable (encapsulate some state); shareable strategies can reduce the problem, but they should not maintain state across invocations

Chapter 16: Template Method

Motivation

Intent and Applicability

Structure and Collaborations

Consequences

Motivation

- A random number *r* with the distribution function *F*(*r*) is generated from a random number *u* with the uniform distribution by the following mapping function:

  $r = F^{-1}(u)$

- Therefore, the algorithm for generating random numbers of arbitrary distribution is:

```
double RandomGenerator::getRndNum () {
  double u = urnd();
  double r = convert(u);
  return r;
}
```

  where `urnd()` is a function that generates a uniform random number, and `convert()` is a polymorphic member function that implements $F^{-1}(u)$.

Motivation

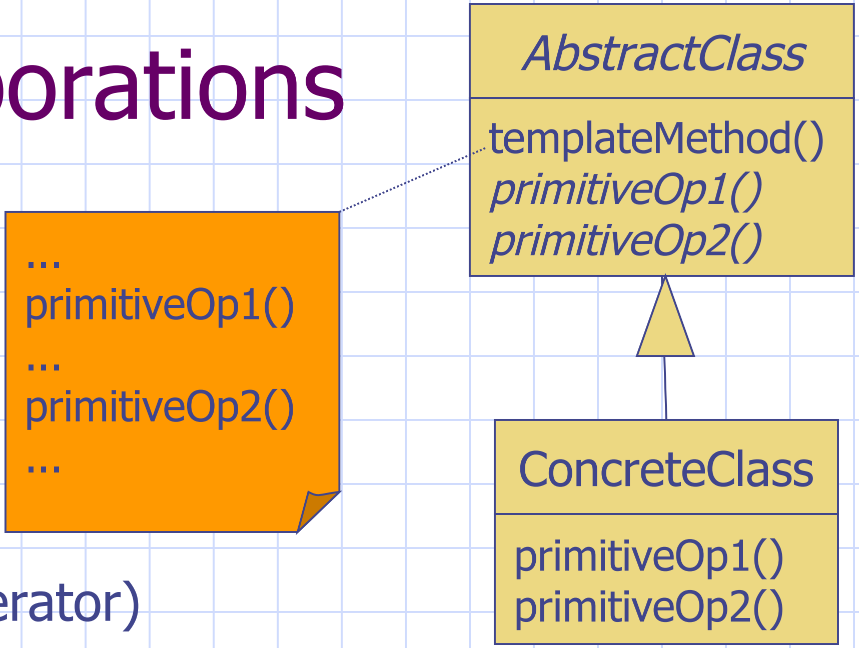
- Consequently, the abstract class `RandomGenerator` can implement the operation `getRndNum()` by defining a fixed algorithm for generating random numbers of arbitrary distribution, while the derived concrete classes will define the step `convert()`.
- Such a method in a base class is called a *template method*. It defines a fixed algorithm in terms of some concrete and some abstract steps that will be specified by derived classes. In other words, a template method fixes the ordering of the steps of an algorithm, while leaving the derived classes to specify some of them.

Intent and Applicability

- Intent (class/behavioral):

  Define the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Applicability: the Template Method DP should be used:
  - to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary
  - when common behavior among subclasses should be factored and localized in a common class to avoid code duplication ("refactoring to generalize" or "algorithmic generalization"); first identify the differences in the existing code of subclasses' methods, then separate the differences into new operations, and finally gather the common parts into a template method that calls these operations
  - to control subclasses extensions; a template method allows subclasses to provide behavior only at specific points

Structure and Collaborations



- Participants:
  - **AbstractClass** (RandomGenerator)
    - defines abstract primitive operations that concrete subclasses redefine to implement steps of an algorithm
    - implements a template method defining the skeleton of an algorithm; the template method calls primitive operations as well as operations defined in AbstractClass or those of other objects
  - **ConcreteClass** (UniformGenerator, ExpGenerator)
    - implements the primitive operations to carry out specific steps
- Collaborations:
  - ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm

Consequences

- Template methods are a fundamental technique for code reuse. They are particularly important in class libraries and frameworks, because they are the means for factoring out common behavior in library classes
- A parent class calls the operations of a subclass, not the other way around ("the Hollywood principle: Don't call us, we'll call you")
- Template methods call the following kinds of operations:
  - concrete operations (of AbstractClass or client classes), which *cannot* be redefined in subclasses
  - abstract operations, which *must* be redefined in subclasses
  - "hook operations," which are polymorphic operations with a default behavior in the base class (often empty), which *can* but need not be redefined in subclasses

It is important for the designers of subclasses to know which operations are of which of these kinds

Consequences

- A subclass can *extend* a parent class operation's behavior by overriding the operation and calling the parent operation explicitly:

```
void DerivedClass::anOperation () {
  // Extended behavior...
  BaseClass::anOperation();
  // Extended behavior...
}
```

- Unfortunately, it is easy to forget to call the inherited operation. It is better to transform such an operation into a template method and then let subclasses override a hook:

```
void BaseClass::anOperation () {
  // Base class behavior...
  hookOperation(); // does nothing in BaseClass
  // Base class behavior...
}
```

Chapter 17: Visitor

Motivation

Intent and Applicability

Structure and Collaborations

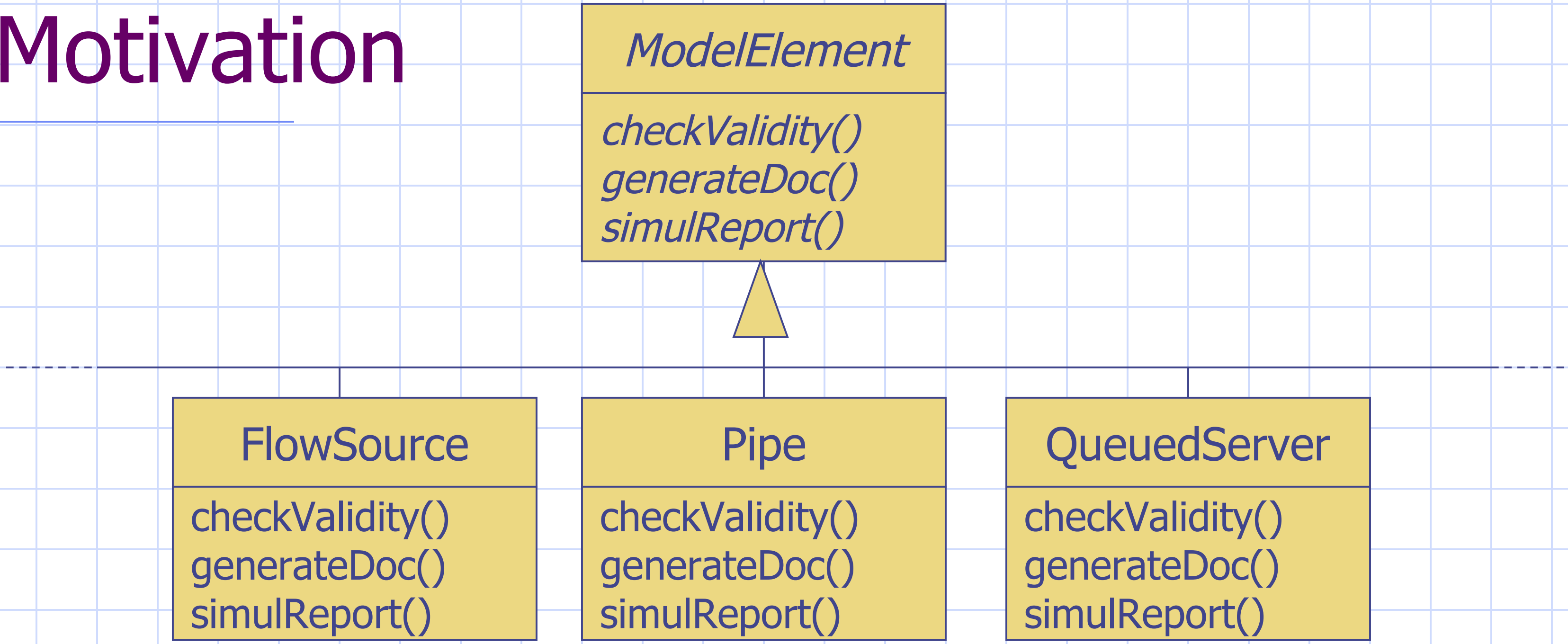
Consequences

Motivation

In TSS, we need to perform different operations on the defined structure of model elements. We want to check the model against violations (e.g., a target assigned to a flow sink or a semaphore without controlled sources), to generate a documentation (a list of model elements with their properties), or to generate a simulation report (with the details about transported vehicles and sizes of queues). Most of these operations will need to treat objects differently, according to their class.

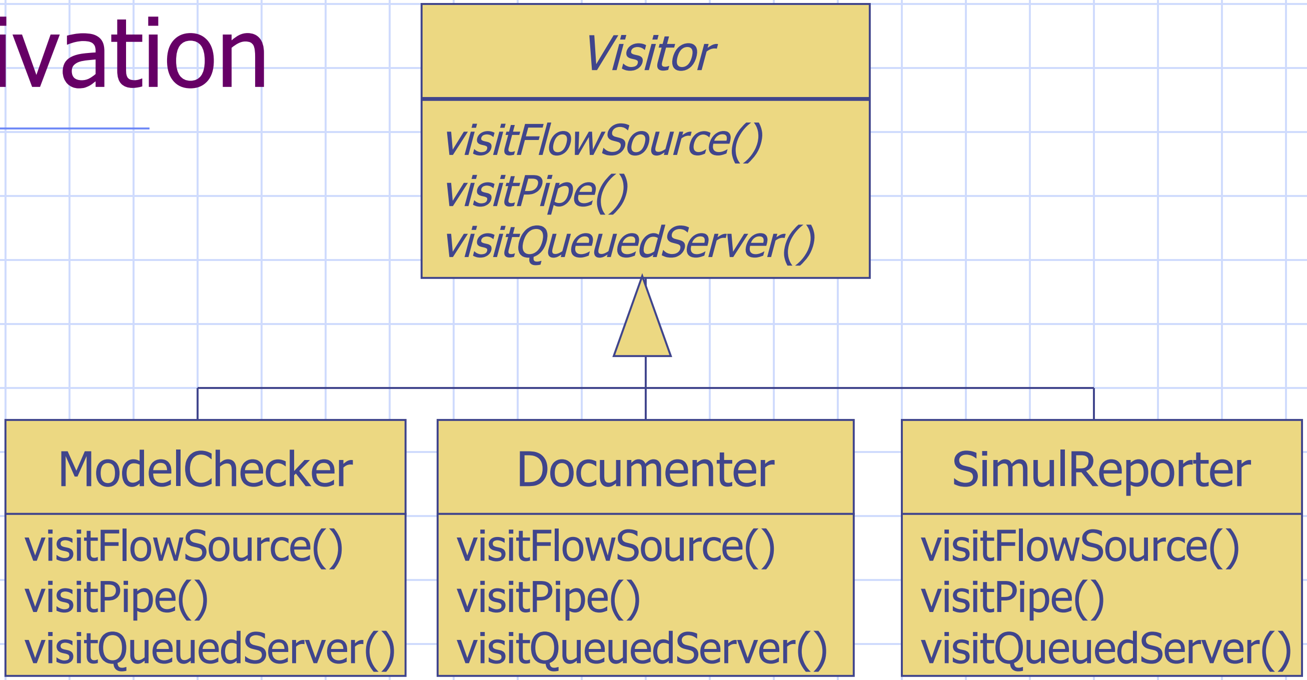
One approach assumes incorporation of these operations into the existing class hierarchy.

Motivation



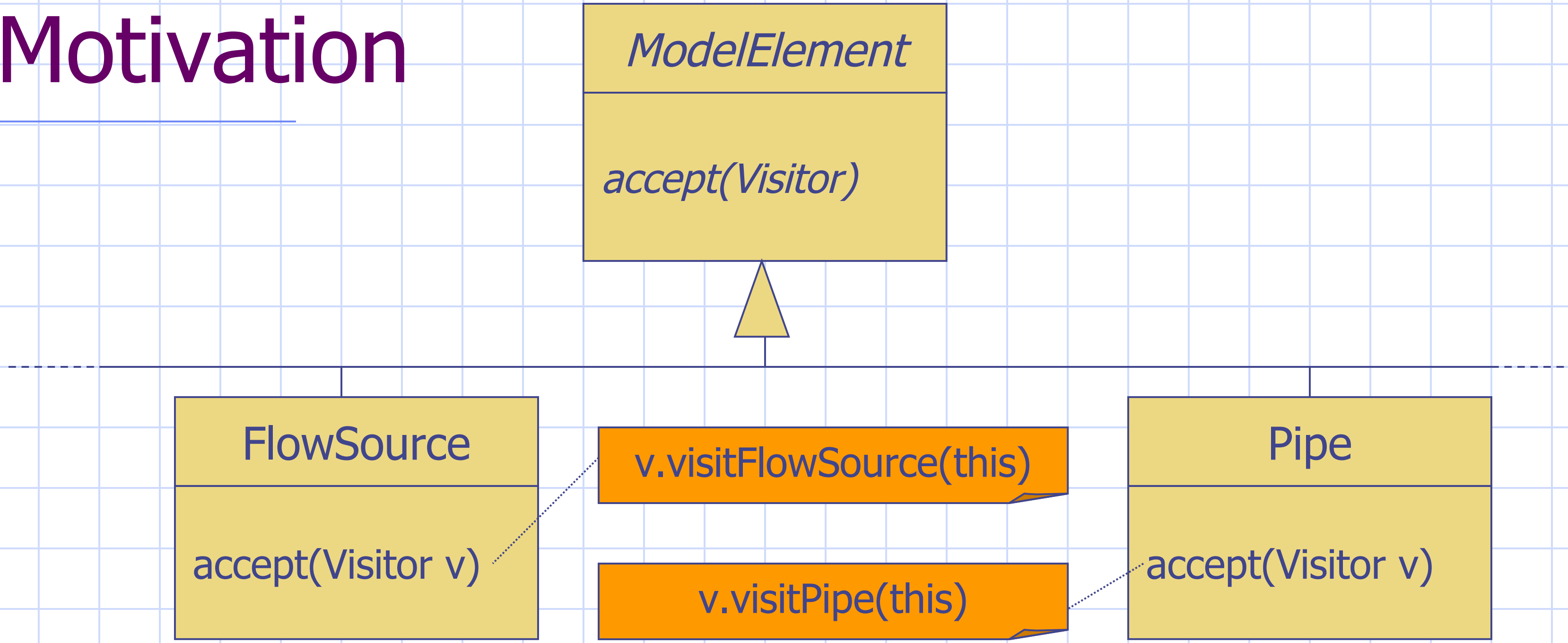
This leads to a system that is hard to understand and maintain, because the main class hierarchy is “spoiled” with the operations and code for different unrelated activities. Adding a new operation usually requires recompilation of the entire hierarchy. It would be better if each new operation could be added separately, and the main hierarchy were independent of the applied operations.

Motivation



This can be achieved by packing related operations from each class in a separate object, called *visitor*, and passing it to elements of the model structure as it is traversed. When an element “accepts” the visitor, it sends a request to the visitor that encodes the element’s class (call-back). It also includes the element as an argument. The visitor will then execute the operation for that element.

Motivation



Ultimately, there will be two class hierarchies: one for the elements being operated on (the ModelElement hierarchy) and one for the visitors that define operations on the elements (the Visitor hierarchy). A new operation is created by adding a new subclass to the visitor hierarchy, without affecting the main hierarchy.

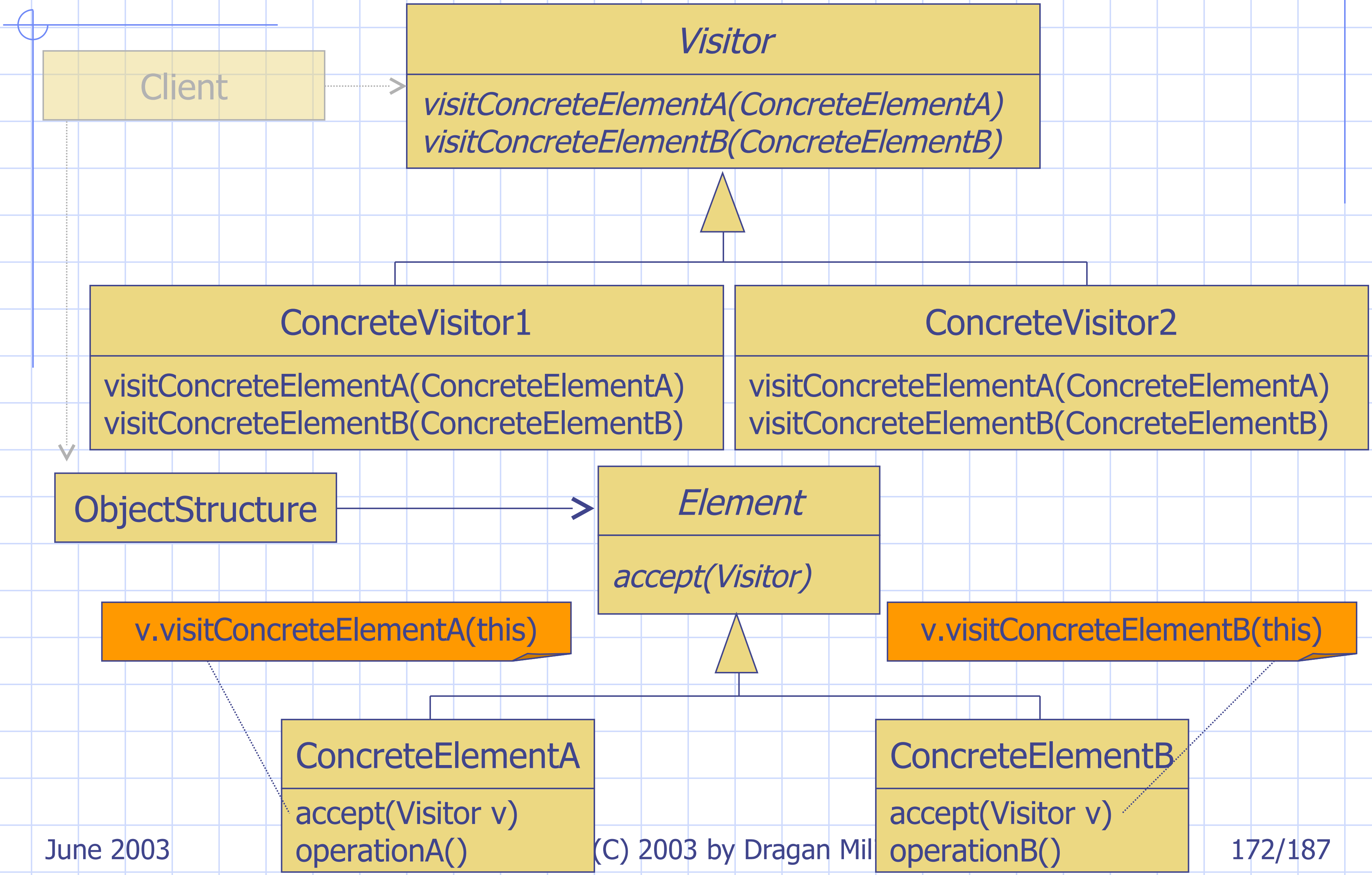
Intent and Applicability

- Intent (object/behavioral):

  Represent an operation to be performed on the elements of an object structure. Visitors lets you define a new operation without changing the classes of the elements on which it operates.

- Applicability: use the Visitor DP when:
  - an object structure contains many classes of objects with different interfaces, and you want to perform operations on these objects that depend on their concrete classes
  - many distinct and unrelated operations need to be performed on objects in a structure, and you want to avoid “polluting” their classes with these operations; Visitor lets you keep related operations together in one class
  - the classes defining the object structure rarely change, but you often want to define new operations over the structure; changing the object structure requires changing interfaces of all visitors

Structure and Collaborations



Client

Visitor

visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

ConcreteVisitor1

visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

ConcreteVisitor2

visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)

ObjectStructure

Element

accept(Visitor)

v.visitConcreteElementA(this)

v.visitConcreteElementB(this)

ConcreteElementA

accept(Visitor v)
operationA()

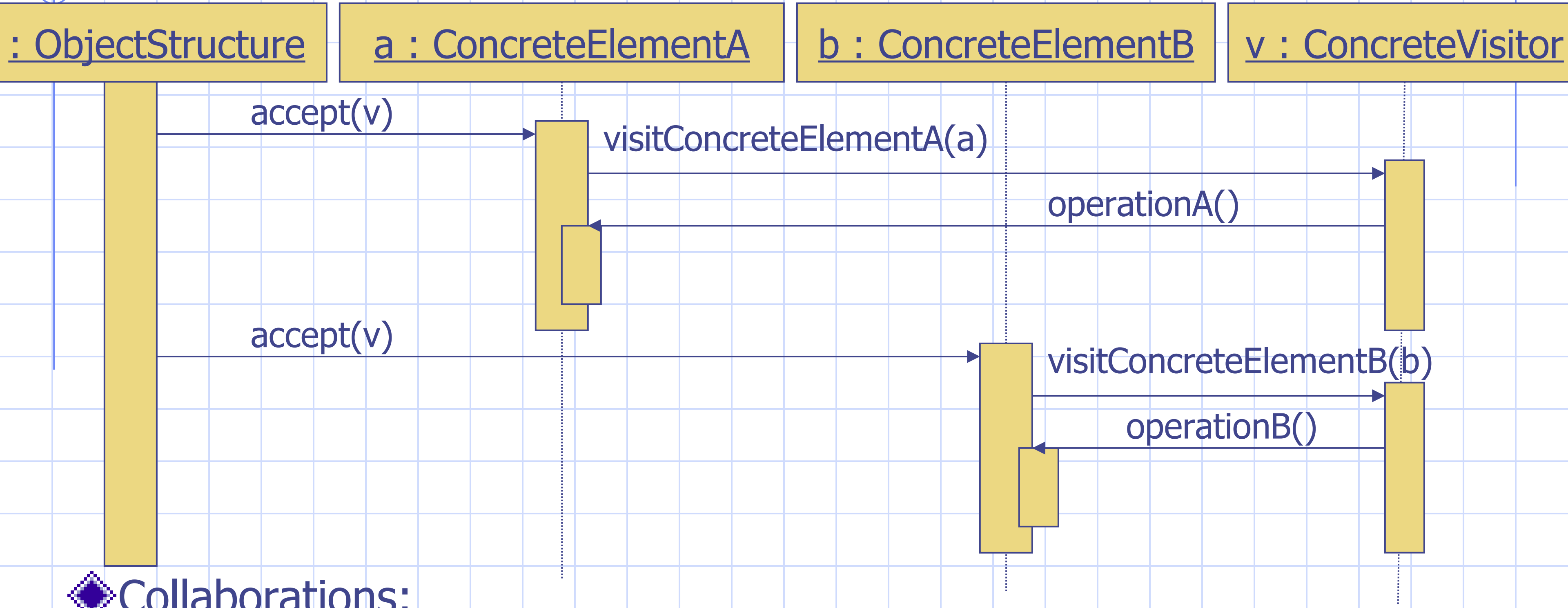
ConcreteElementB

accept(Visitor v)
operationB()

Structure and Collaborations

- Participants:
  - **Visitor** (Visitor)
    - declares a visit operation for each class of ConcreteElement in the object structure; the operation's name and signature identifies the class that sends the visit request to the visitor; that lets the visitor determine the concrete class of the element being visited; then the visitor can access the element directly through its particular interface
  - **ConcreteVisitor** (ModelChecker, Documenter, SimulReporter)
    - implements each operation declared by Visitor; each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure; ConcreteVisitor provides the context for the algorithm and stores its local state; this state often accumulates results during the traversal of the structure
  - **Element** (ModelElement)
    - defines an accept operation that takes a Visitor as an argument
  - **ConcreteElement** (FlowSource, Pipe, QueuedServer, etc.)
    - implements an accept operation that takes a Visitor as an argument and calls back the operation of the visitor that corresponds to its class

Structure and Collaborations



Collaborations:

- A client must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor
- When an element is visited, it calls the Visitor operation that corresponds to its class, passing itself as an argument to let the visitor access its state, if necessary

Consequences

- Visitor makes adding new operations easy. A new operation is added simply by adding a new visitor, instead of spreading the functionality over many classes
- A visitor gathers related operations and separate unrelated ones. Any algorithm-specific data structures can be hidden in a visitor
- Adding new ConcreteElement classes is hard, because a new abstract operation is needed in the Visitor class, and a corresponding implementation in ConcreteVisitor. Sometimes a default implementation (often empty) can be provided in Visitor to reduce the overhead. If it is more likely to change the algorithm applied over an object structure, and the object structure is stable, it is useful to apply the Visitor DP. Otherwise, it is easier to define operations in the classes that make up the structure

Consequences

- Visitors can accumulate state as they visit each element in the object structure. This is better than accumulating the state in arguments of operations, global objects, or objects in the structure
- Visitor assumes that the ConcreteElement interface is powerful enough to let visitors do their job. As a result, it often forces to provide public operations to access the element’s internal state, possibly compromising its encapsulation

Chapter 18: Composite

Motivation

Intent and Applicability

Structure and Collaborations

Consequences

Examples of Usage

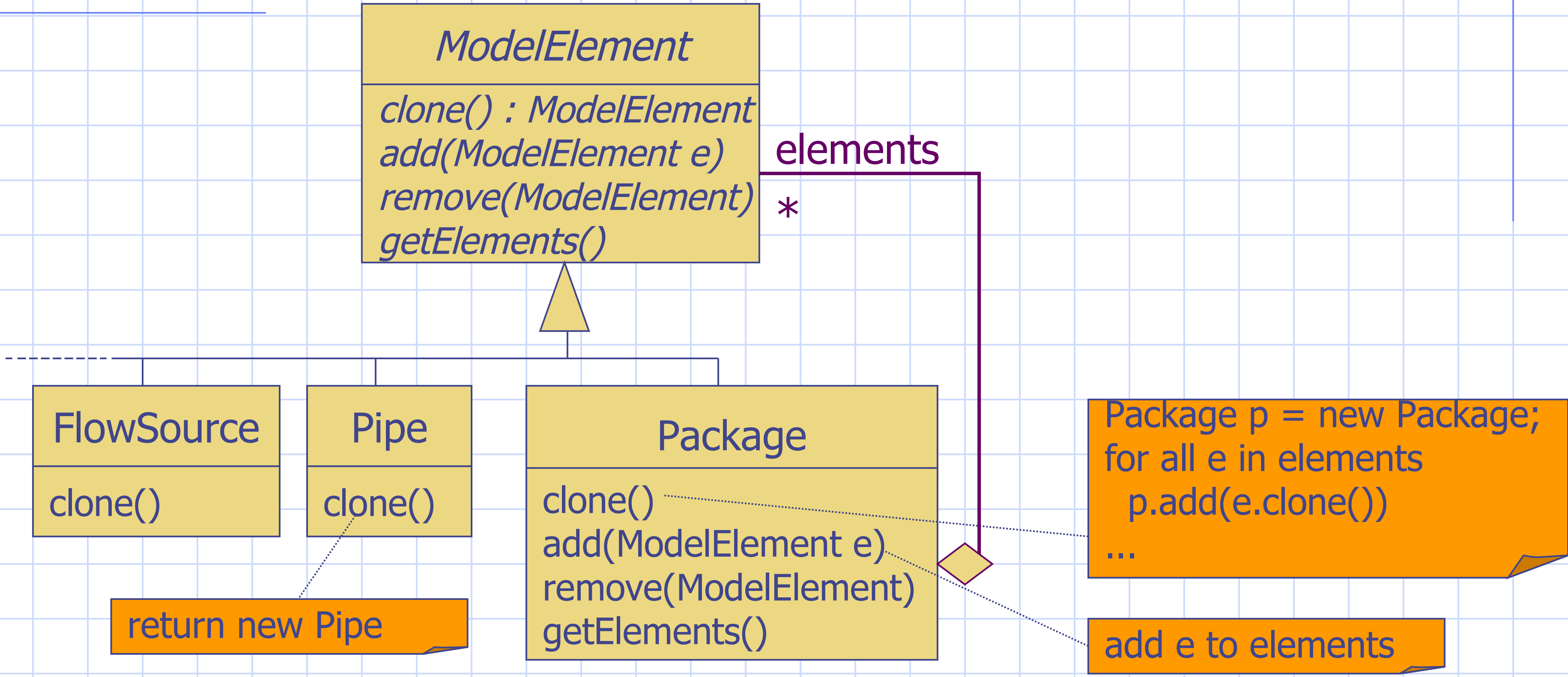
Motivation

A TSS model may consist of many different elements, e.g. flow sources and sinks, pipes, servers, and semaphores. It may become very clumsy to manipulate with and organize if all of its elements are put in the model as a bag of unordered elements.

A possible solution may be to introduce a concept of a *package* as a general grouping mechanism. A package may group all other model elements, as well as other nested packages. This way, we may organize the elements into a tree-shaped hierarchy of packages and other primitive elements.

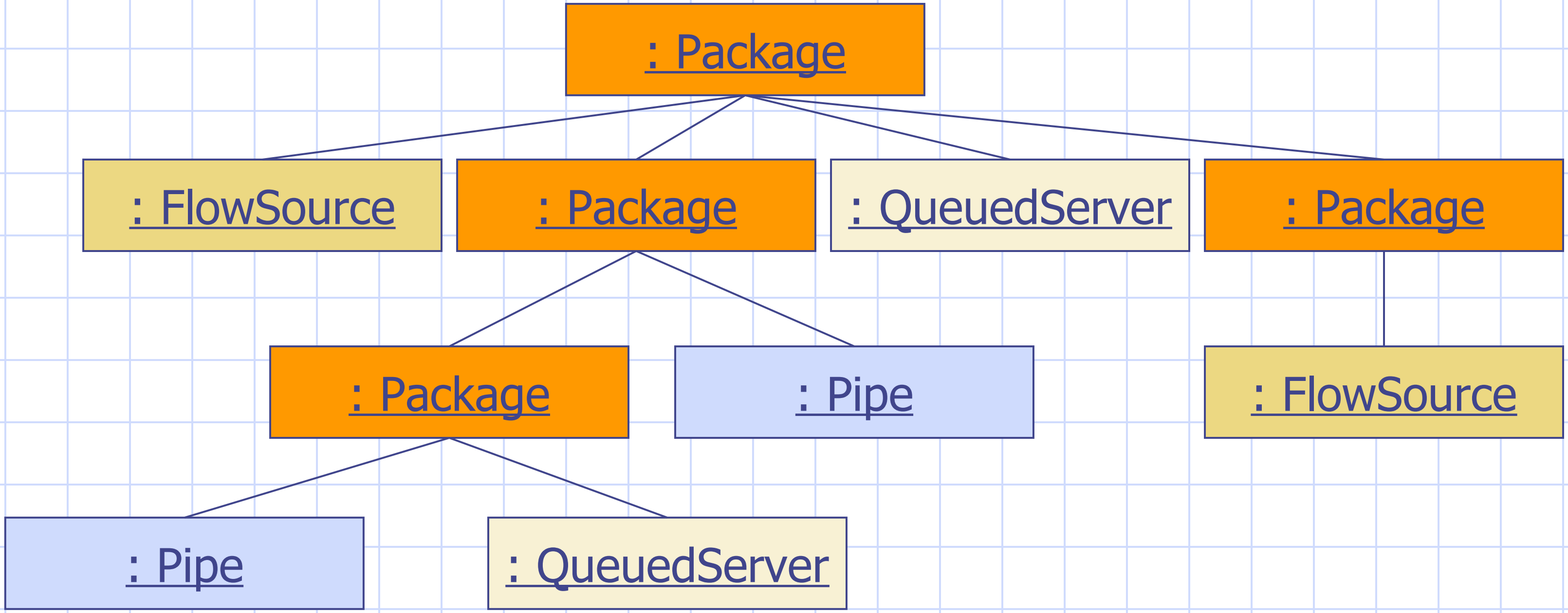
However, we may need to treat packages and other elements uniformly. We may want to visit them by visitors, or we may want to copy/paste an element (to clone it). If we keep the Package class unrelated with the others, we will have the client code more complex because it will treat the elements and packages differently, even if it doesn’t want to.

Motivation



The Composite DP offers a solution. The key is that the abstract class ModelElement provides the interface and represents *both* primitives (e.g. source, pipe, etc.) and their compositions (package).

Motivation



It describes how to use recursive composition of objects, allowing tree-shaped hierarchical structures, whereby leaf nodes are primitives, and non-leaf nodes are composites. It makes the client's code simple because it may treat all nodes equally (as generalized elements).

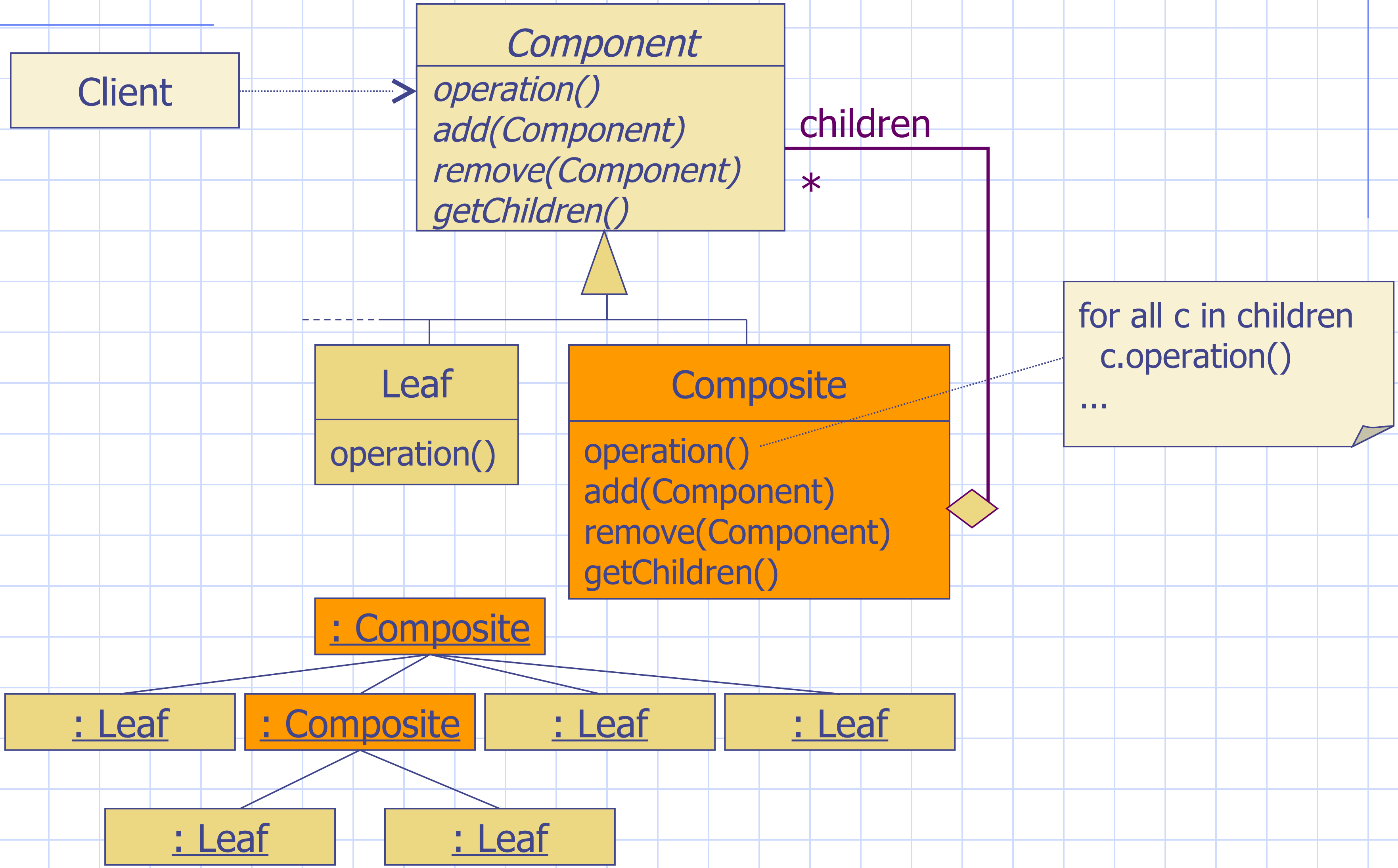
Intent and Applicability

- Intent (object/structural):

  Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- Applicability: use the Composite DP when you want:
  - to represent part-whole hierarchies of objects
  - clients to be able to ignore the difference between composition of objects and individual objects; clients will treat all objects in the hierarchical structure uniformly

Structure and Collaborations



Client

Component

operation()
add(Component)
remove(Component)
getChildren()

children

*

Leaf

operation()

Composite

operation()
add(Component)
remove(Component)
getChildren()

for all c in children
  c.operation()
...

: Composite

: Leaf

: Composite

: Leaf

: Leaf

: Leaf

: Leaf

Structure and Collaborations

- Participants:
  - **Component** (ModelElement)
    - declares the interface for objects in the composition
    - implements default behavior for the interface common to all classes
    - (optional) declares an interface for accessing and managing its child and parent components in the recursive structure and implements it if appropriate
  - **Leaf** (FlowSource, Pipe, etc.)
    - represents leaf objects in the composition; a leaf has no children
    - defines behavior for primitive objects in the composition
  - **Composite** (Package)
    - defines behavior for components having children
    - stores child components
    - implements child-related operations in the Component interface
  - **Client**
    - manipulates objects in the composition through the Component interface

Structure and Collaborations

- Collaborations:
  - Clients use the Component class interface to interact with objects in the composite structure; if the recipient is a Leaf, the request is handled directly; if the recipient is a Composite, then it usually forwards request to its child components, possibly performing additional operations before and/or after forwarding (recursion through polymorphism)

```
Client:
  theRoot.operation()

Composite::operation  () {
  for each c in children c.operation()
}

Leaf::operation () {
  do specific actions
}
```

Consequences

- The Composite DP:
  - defines recursive object hierarchies of primitive and composite objects, whereby primitive objects can be composed into more complex objects, which in turn can be composed, etc. recursively
  - makes the clients simple, because they can treat composite structures and individuals uniformly; they usually don’t know (and shouldn’t care) whether they are dealing with a leaf or a composite (avoidance of if-then and case structures in code)
  - makes it easier to add new kinds of components; newly defined Composite or Leaf subclasses work automatically with existing structures and clients
  - can make the design overly general; sometimes it is necessary to restrict the types of components in a Composite; then you must rely on the underlying type-detection system to enforce the constraints

Examples of Usage

- Graphical editors:
  - a graphic (abstract component)
  - concrete graphics: rectangles, lines, circles, etc. (leaves)
  - a grouped graphic (composite)
- File system:
  - a file-system element (abstract component)
  - a file (leaf)
  - a folder (composite)
- UML metamodel:
  - a model element (abstract component)
  - concrete model elements: class, attribute, association, etc. (leaves)
  - a package (composite)

Part IV: Conclusions

An Example from Your Domain?

Summary

What’s Next?

Questions and Answers

Discussion

Evaluation