INTRODUCTION TO UML STRUCTURAL MODELING

In the last set of UML slides our focus was on modeling behavior. Behavior models are very much ***dynamic*** - they show the system as it changes, perhaps by following its nature process flow, perhaps by its reactions to events. In contrast, we will now look at structural modeling. We want to represent the ***static*** structure of a system – how its components are interconnected, and composed.

Language Elements

Class Diagrams

Component Diagrams

BASIC MODELING ELEMENTS

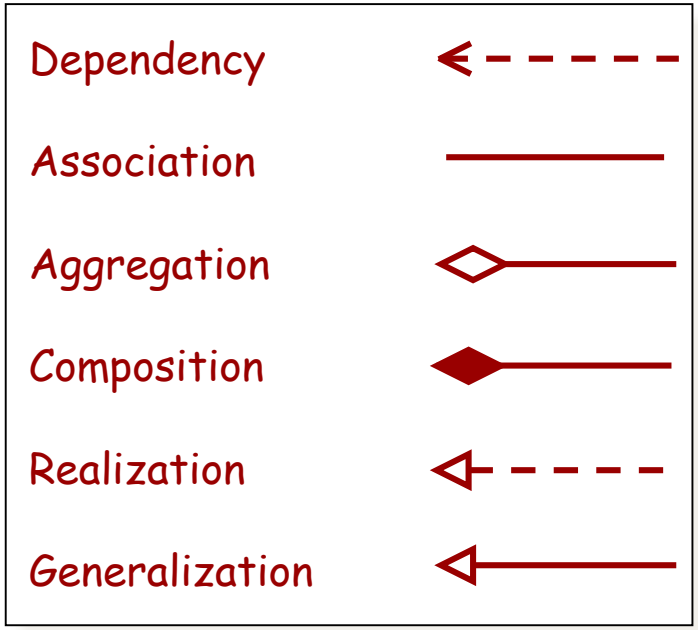
Let's start by building a common vocabulary of the elements in structural modeling. Many of these you will have already seen – however, you will have seen them in terms of code. That's not always the case though. Try to think more generically, in terms of relationships between concepts. This will be more useful, since we will see this notation in more than class diagrams.

- UML diagrams contain a set of basic modeling elements
  - These represent the primitive elements for UML.
- Modeling elements include:
  - Object
  - Class
  - Package
  - Component, Interface
  - Relationships (right)

More on these terms later.

coupling

A side note: coupling is the degree to which two components of a system depend on one another. A high coupling indicates two components have a strong relationship - this can be problematic since change in one component is then more likely to impact the second component. Hence, coupling is something to avoid.

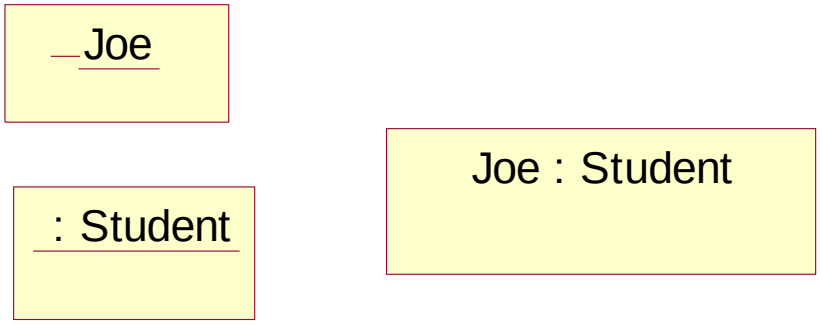


Now we'll start going over each of the structural UML elements, starting with objects.

OBJECT

- An abstraction that has state, behavior, and identity
  - *"An object represents a particular instance of a class. It has identity and attribute values"* - UML definition
- Represented as a rectangle with optional object name and class name all underlined
  - An optional second compartment shows relevant type information

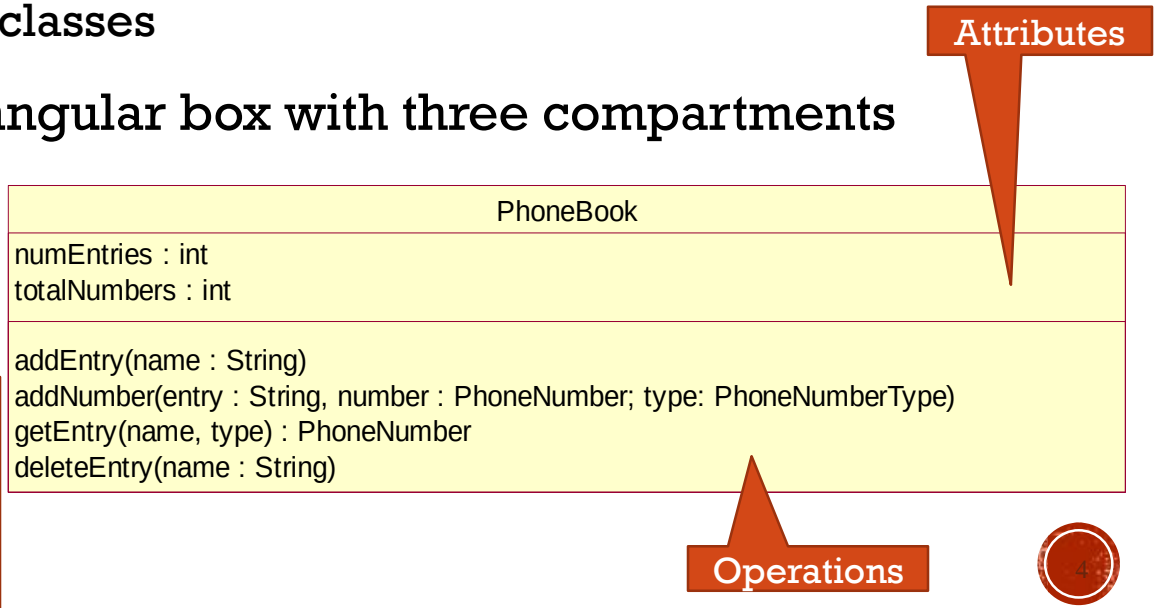
For this example, we have UML notation that indicates (left to right): an object called Joe, an object of type Student, and an object called Joe of type Student.



CLASS

- *"A class is the descriptor for a set of objects with similar structure, behavior, and relationships"*
- Represents a concept within the system being modeled
- Defines object's features
  - Data structure (attributes)
  - Behavior (operations)
  - Relationships to other classes
- Represented as a rectangular box with three compartments
  - Class name
  - Data features
  - Behavioral features

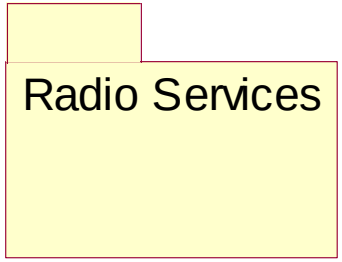
A feature is simply something that defines a class. Here we use the terms "data features" and "behavioral features" to be generic - for a class diagram (right), it boils down to variables and methods.



PACKAGE

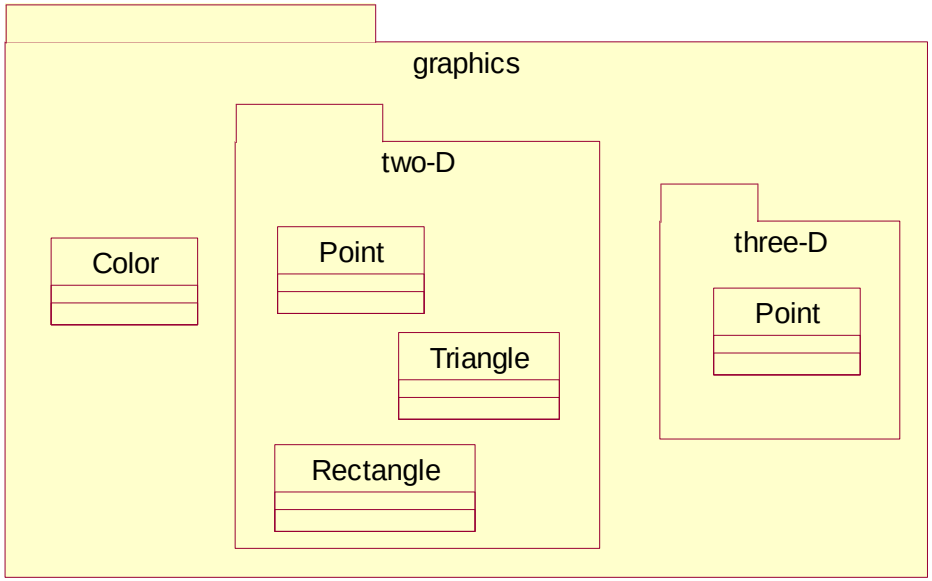
For us packages provide a nesting (recursive) way to structure elements in a diagram. This provides not only structure but also abstraction, since one can try to think in terms of the package instead of having to track its individual elements.

- "A package is a grouping of model elements"
  - May contain any types of model elements, including nested packages
- Package owns the elements they contain
  - All elements belong to a single package
  - Elements within a package may reference elements from other packages
- Represented as a folder icon



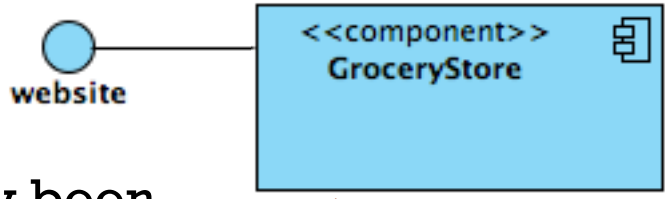
Packages define a naming context for their elements

- Elements with same name can exist in the model (risky!)
- The diagram on right contains 2 Point classes:
  - `graphics::two-D::Point` and `graphics::three-D::Point`



COMPONENTS AND INTERFACES

- *"A component represents a modular, reuseable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces"*
  - A component typically represents a software element in the system (.c, .h, .java, .dll, .exe, .so, etc.)



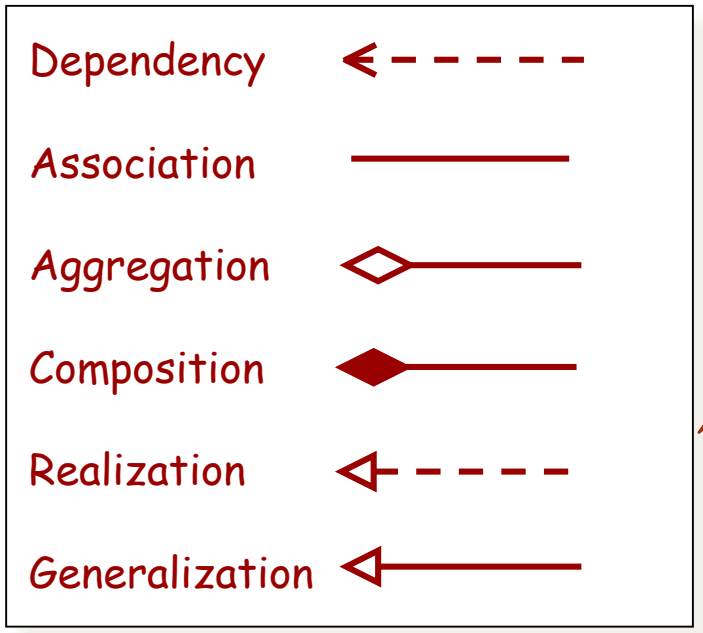
The UML notion of a component has historically been somewhat different than the community's

- Others think of a component as a design-time concept
  - Versus the notion of a "component as a bigger object"
  - UML historically said to use Packages for this
- The "classic" concept and notation is ball and socket diagrams (*stay tuned for an example*)

This diagram indicates that the *GroceryStore* component provides the *website* interface. The little icon in the upper right of the *GroceryStore* element is a symbol meaning component.

RELATIONSHIPS

- "A semantic connection among model elements"
  - Represents communication paths, references, and other connections among modeling elements.
- UML relationships:

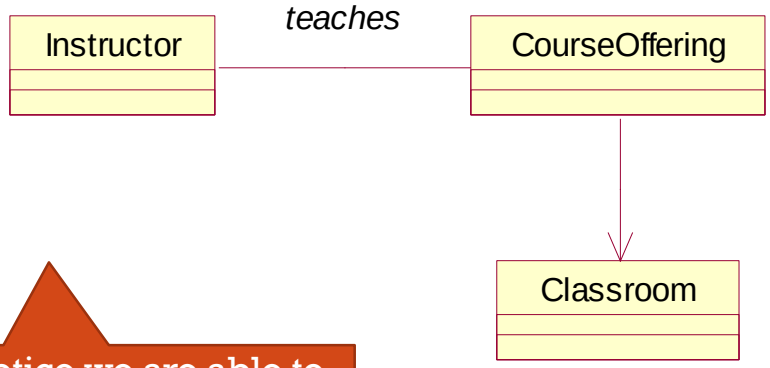


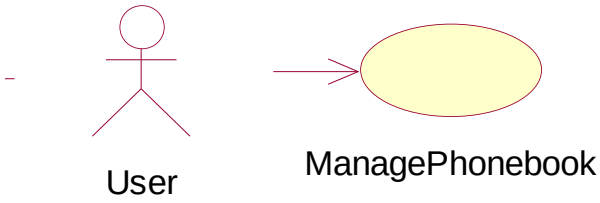
Note: these slides define the UML relationships with words, not code. While this is useful since it stays generic, it may lose the concreteness that helps you understand it. So, if these slides seem too abstract, take a look in BlackBoard where these slides were downloaded. There you will find a source file containing code samples of these relationships.

ASSOCIATIONS

This is the appearance of an association relationship: a line.

- "The semantic relationship between two or more classifiers that specifies connections among their instances"
- A structural, permanent (static) relationship
- Represented as a line with optional arrowhead
  - Arrowhead indicates unidirectional; no arrow indicates bi-directional
  - May optionally be named

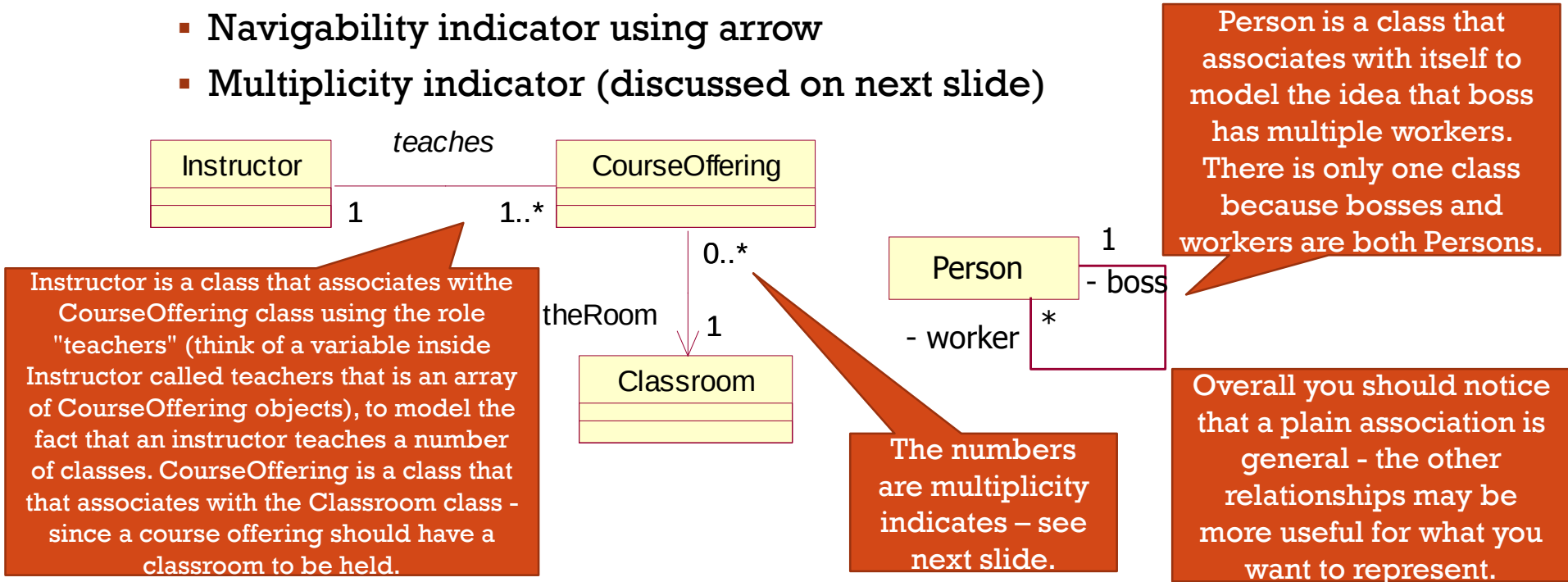




Notice we are able to use this relationship in both Use Case Diagrams (SER215), and Class Diagrams.

ASSOCIATION ENDS

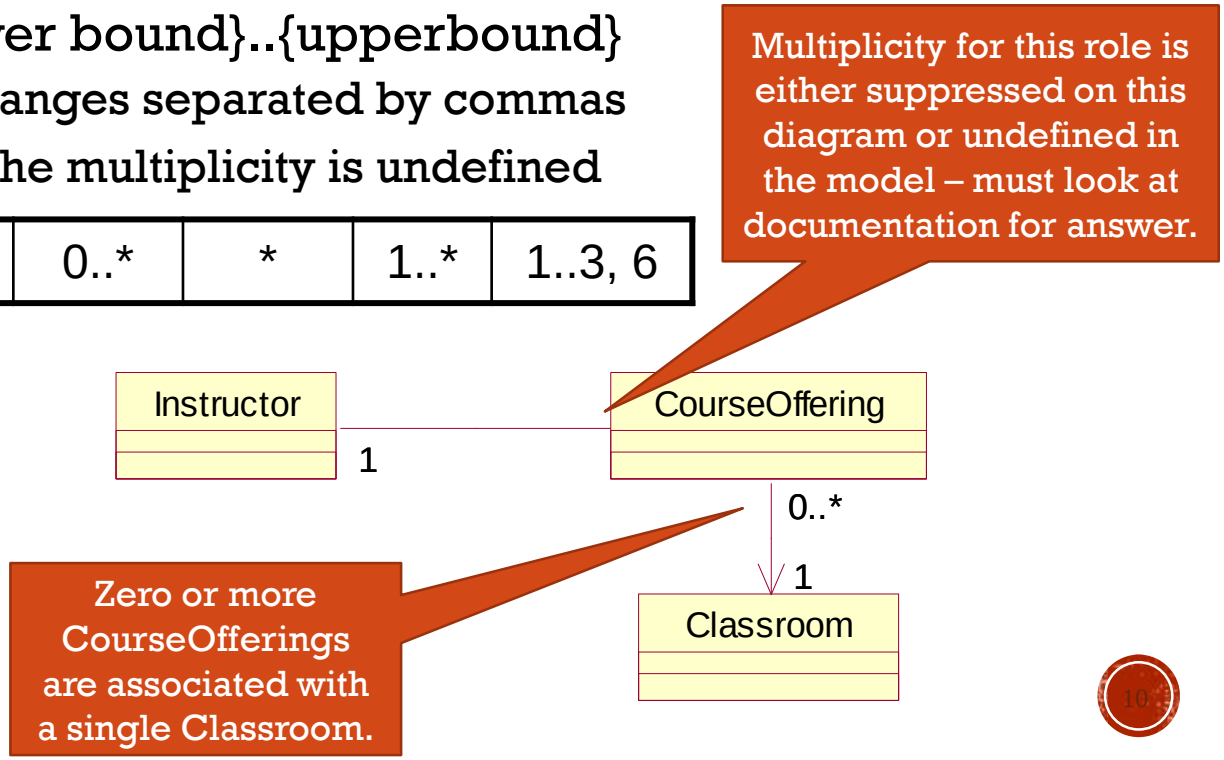
- The end of an association indicates the connection of an association with a class. Above, A connects to B.
- May contain any of the following properties:
  - Role name
  - Navigability indicator using arrow
  - Multiplicity indicator (discussed on next slide)



MULTIPLICITY

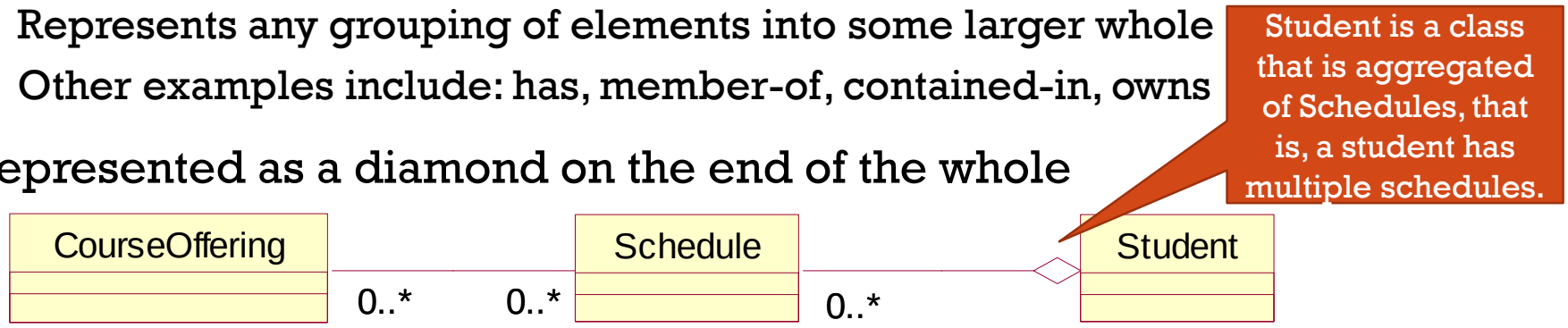
- "The range of allowable cardinalities a set may assume"
  - Applies to roles in an association, but also to other parts of UML (e.g. parts within composites, repetitions, etc.)
- Represented as {lower bound}..{upperbound}
  - Can have multiple ranges separated by commas
  - No value indicates the multiplicity is undefined
  - Examples:

1	0..*	*	1..*	1..3, 6



AGGREGATION AND COMPOSITION

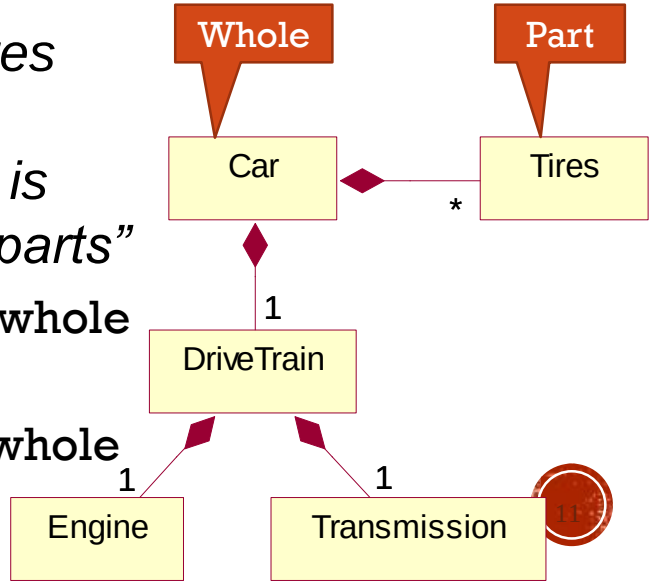
- "A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part"
  - Represents any grouping of elements into some larger whole
  - Other examples include: has, member-of, contained-in, owns
- Represented as a diamond on the end of the whole



"Composition is a form of aggregation which requires that a part instance be included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts"

- Exclusive Ownership – parts belong to at most one whole (multiplicity on the whole side is 0..1 or 1)
- Lifetime: part's existence depends on existence of whole

Represented as a filled diamond

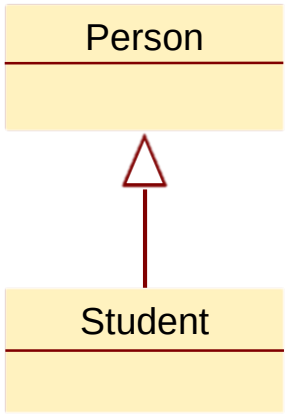


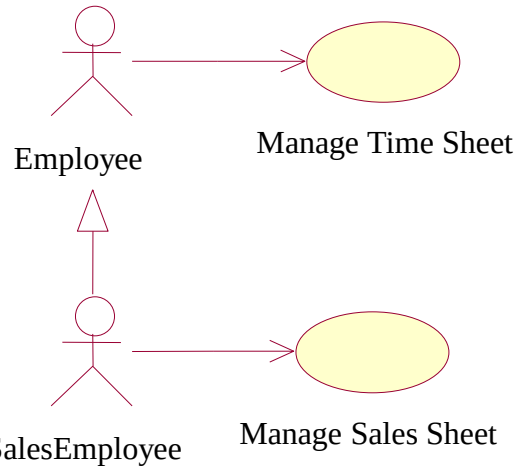
GENERALIZATION

- *"A taxonomic relationship between a more general element and a more specific element"*

These are the ideas from CST200!

- The more specific element
  - Is fully consistent with the more general element (Inheritance)
  - May override the behavior inherited from parent (Overriding)
  - Can add additional features (Extension)
  - May be used anywhere the more general element is allowed (Substitution)
- Represented as a line with a large hollow triangle pointing to the more general element (the parent)



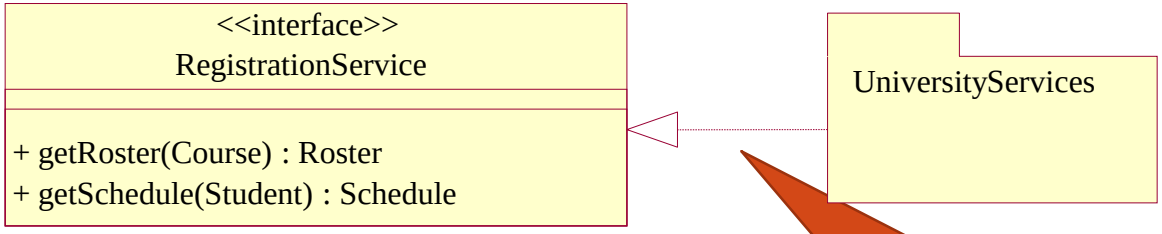


REALIZATION

Same idea as interfaces from CST200.

- "A specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other representing an implementation of the latter (the client)"

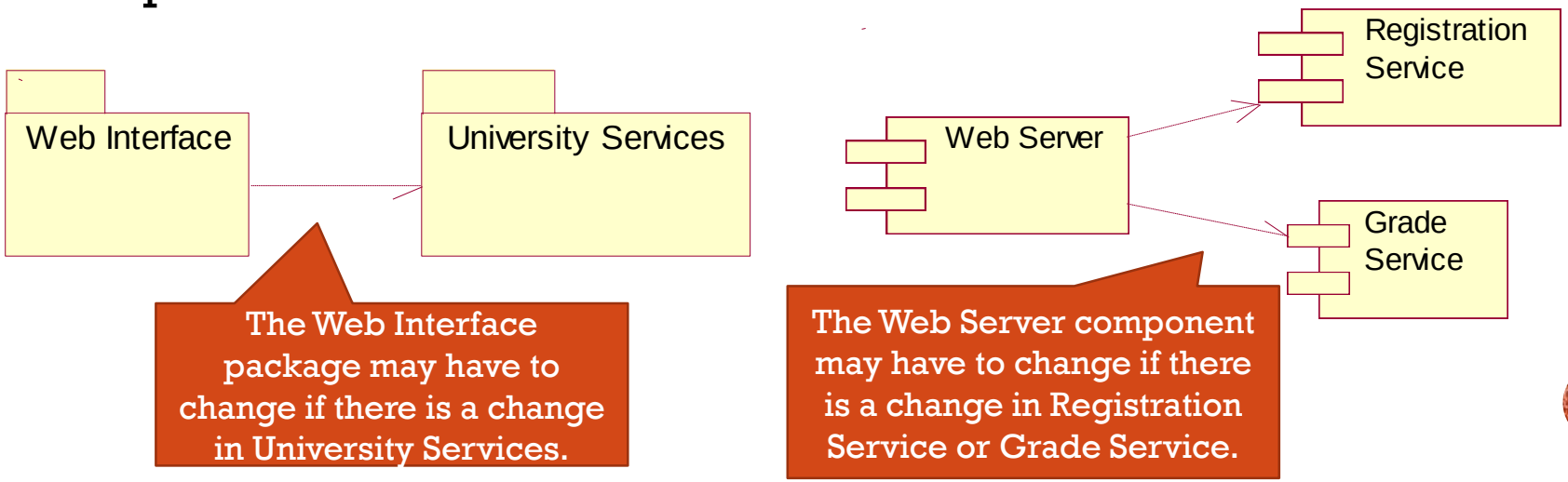
- Represented as a dashed line with a closed, hollow triangle pointing to the specification element (e.g., interface)



UniversityServices is a package that provides concrete functionality for the behaviors defined in the RegistrationService interface.

DEPENDENCY <----

- *"a semantic relationship between two model elements in which a change to the target element may require a change to the source element in the dependency"*
- Reflects a non-structural relationship (vs. a permanent structural association relationship)
- Represented as a dashed arrow between two elements



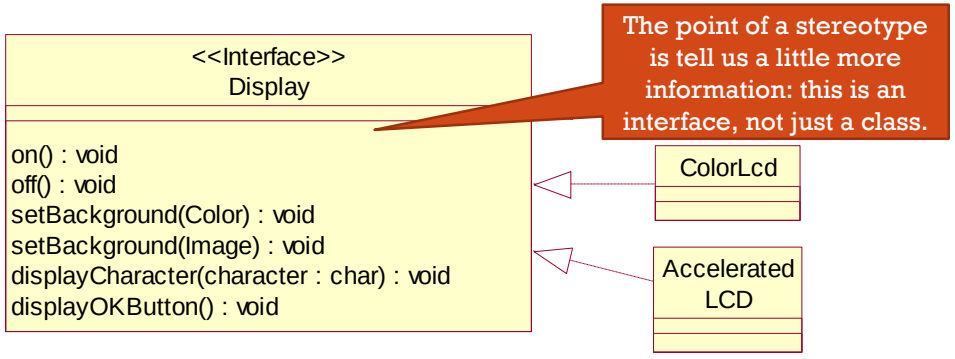
General Extension Mechanisms

- Ways to let us customize and extend semantics of a model.
- May be user-defined, defined in UML specification or within UML profile
  - User-defined semantics depend on modeler and are not defined by UML
- Mechanisms: *Stereotype*, *Constraint*, *Comment*

Stereotype

"…represents a subclass of an existing [sic] element with the same form (attributes & relationships) but with a different intent … a usage distinction … may have additional constraints"

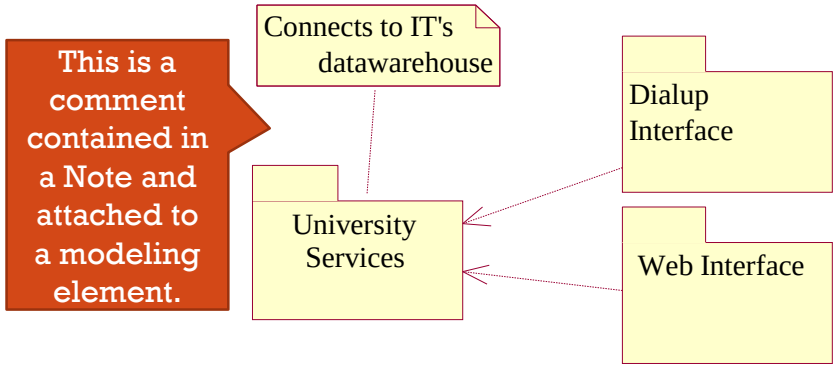
- Stereotyped elements may be labeled as <<stereotype>>
- Can apply to any UML modeling element
- Icons may be used to visually represent the stereotype



Comment

"An annotation attached to an element or a collection of elements."

- Contained in a Note element and attached to 1 or more modeling elements



Now that we have gone over the general "vocabulary" of UML structure, we want to look at specific examples: Class Diagrams, and Component Diagrams. You've likely seen Class Diagrams before (to illustrate a program in a OOP language), but Component Diagrams will be new.

16 STRUCTRAL DIAGRAMS

Class Diagrams

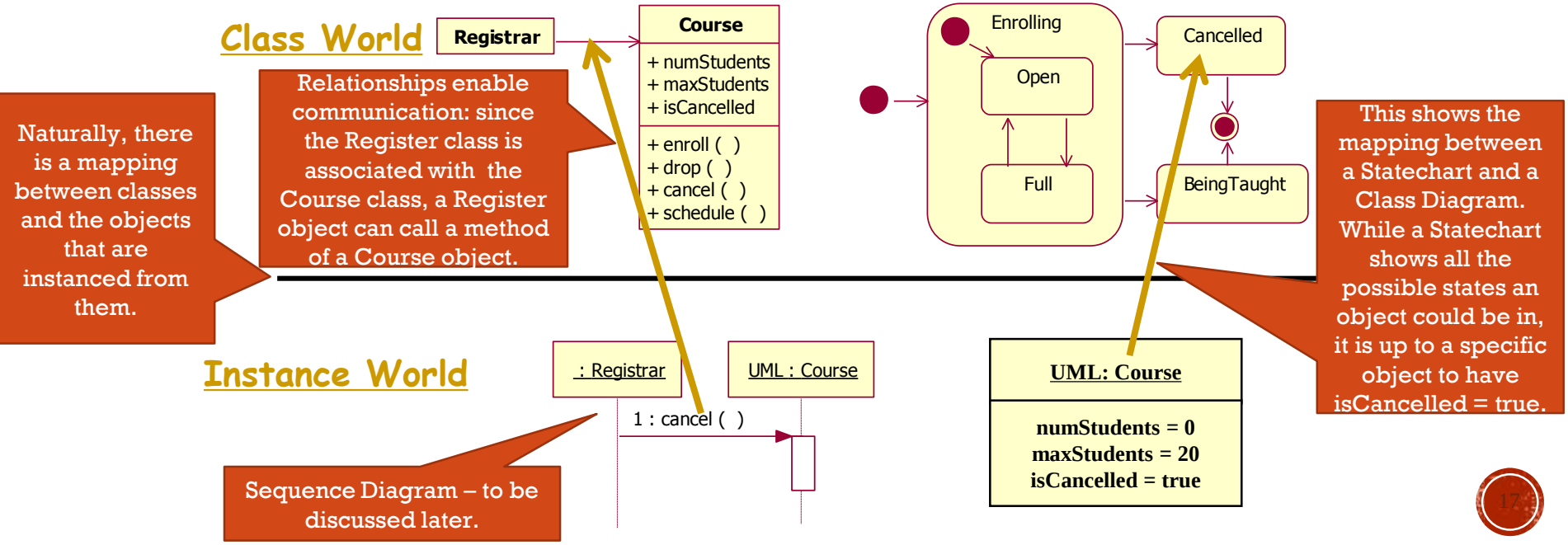
Component Diagrams

Note that while our commentary will have a slight Java flavor to it, these ideas apply to many languages – the keywords just change.

OOA&D: OBJECT COLLABORATIONS

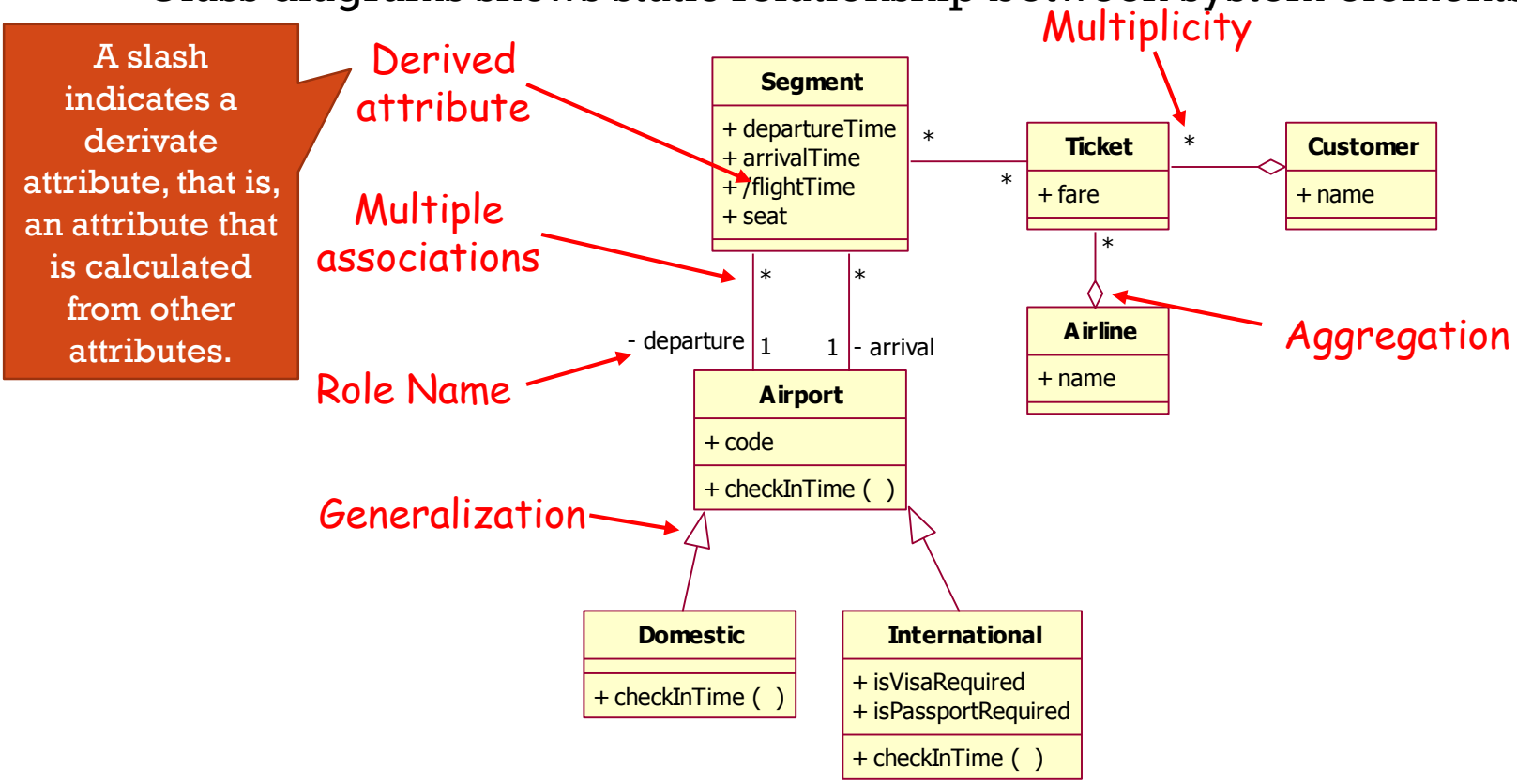
Like calling a method.

- System behavior is realized by objects sending "messages".
  - Associations provide pathways for messages
  - Used to inquire about and/or change an object's state
- Structural diagrams try to specify the world of **classes** - how the program is statically structured. In contrast, behavior diagrams specify the world of **instances** (objects) - how the program acts during runtime.



CLASS DIAGRAMS

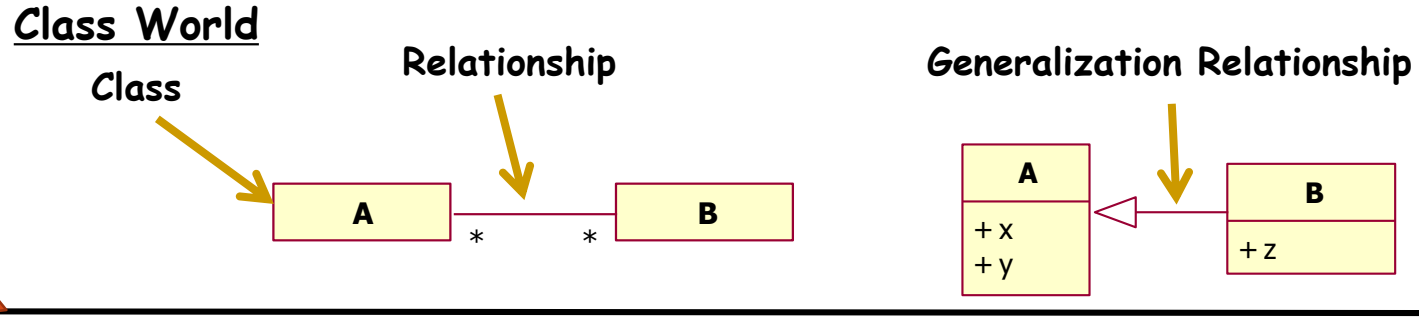
- "A class diagram is a graph of Class(ifier) elements connected by their various relationships"
  - May also contain interfaces, packages, other relationships, objects, etc.
  - AKA Static Structure Diagram
- Class diagrams shows static relationship between system elements

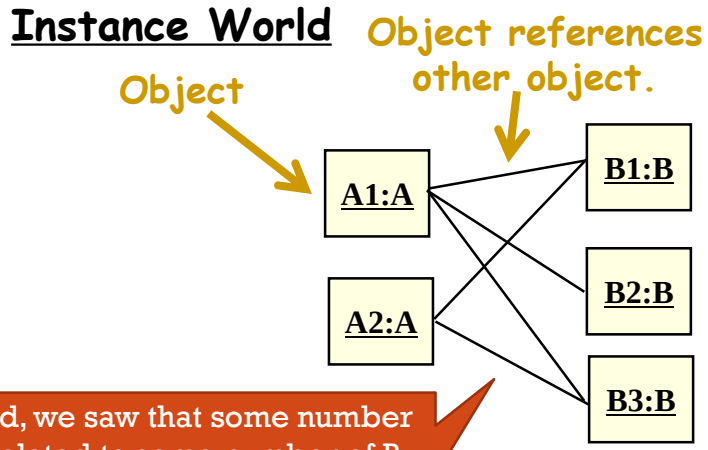


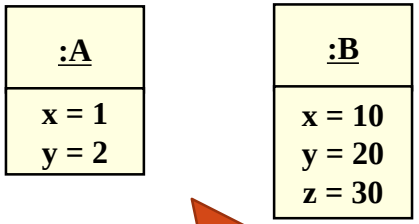
CLASSES/OBJECTS, ASSOCIATIONS/LINKS

- Classes are *descriptors* for many possible objects
- Associations are *descriptors* for many possible links

Again, there is a mapping between classes and the objects that are instanced from them.





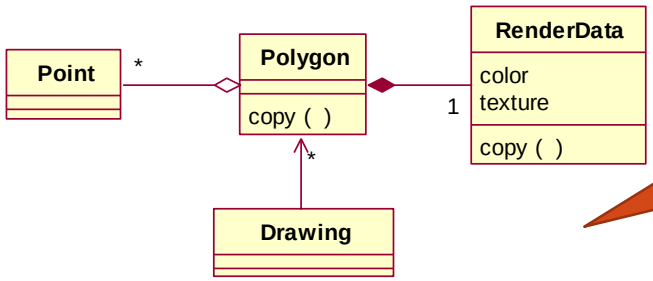


In the class world, we saw that some number of A classes is related to some number of B classes. In terms of a objects (instances), this might mean that A1 is linked to B1 and B3, while B1 is linked to A1 and A2.

Generalizations do not relate instances, instead they specific how objects are built (i.e., an object of type B should have the attributes of class B *and* A).

CLASS DIAGRAM RELATIONSHIPS

- When to use Association, Aggregation, or Composition?
  - No general rule, but there are heuristics:
    - Association is the most general relationship and most common
    - Composition requires parts belong to at most one whole at a time and is responsible for creation and disposition of parts
      - Propagating behaviors implies a composition relationship
    - Sharable aggregation does not imply propagation semantics

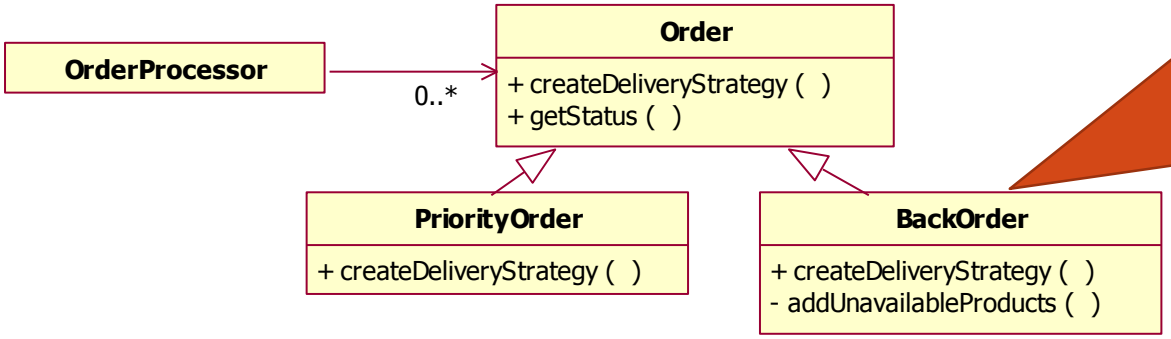


The polygon class aggregates a number of points, but it does not express sole ownership over them. It has complete ownership of a single RenderData. The Drawing class is associated with a number of Polygons.

POLYMORPHIC RELATIONSHIPS

From CST200. The idea is that we can mix and match behavior between parent classes and their children.

- UML's generalization and realization support polymorphism

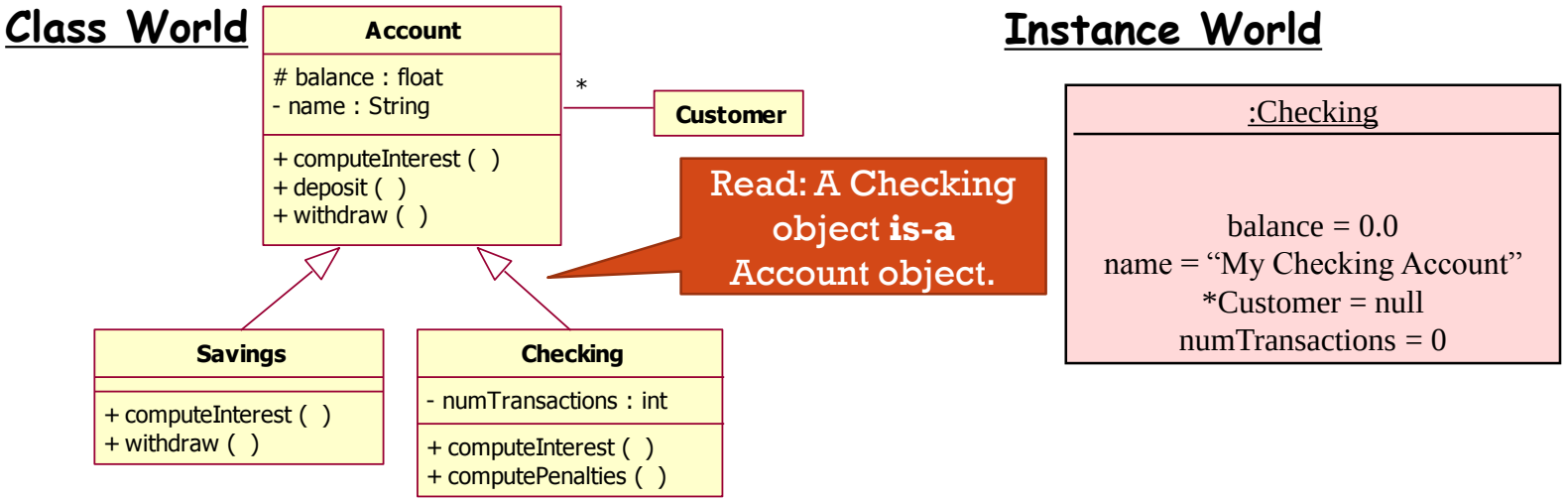


BackOrder defines its own implementation for createDeliveryStrategy, that replaces what the parent had. If we have an object of type BackOrder, the child's newer implementation will be used. Also, if the parent's code for getStatus happened to used createDeliveryStrategy, it would instead use the one provided by the child.

- Provide two salient design features:
  - Reuse (generalization only)
    - Behavior in one class inherited by subclasses
  - Substitution (generalization and realization)
    - Behavior can be added to system without changing existing portions
    - The real advantage to polymorphism

GENERALIZATION

- An inheritance relationship between concrete classes
  - All attributes, operations, and relationships are inherited

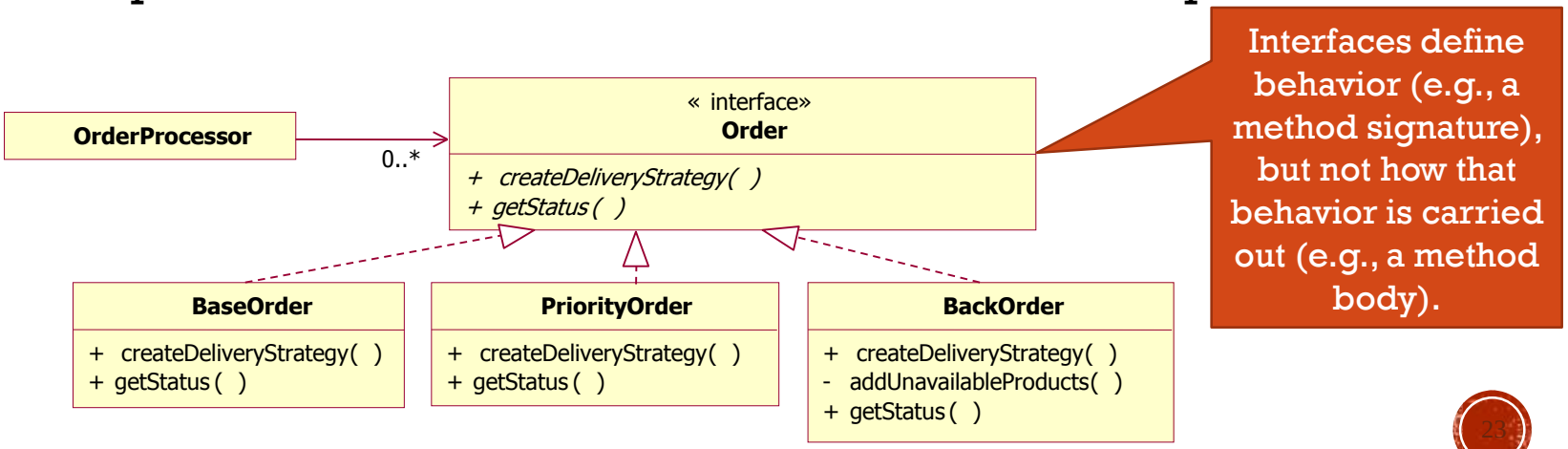


Note: Generalization results in tight coupling between classes

- Child classes inherit *implementation* from parent; changing parent's behavior changes child's behavior – be careful!
- Violates principle of encapsulation

REALIZATION

- Polymorphic relationship between an interface and other classes and interfaces
  - Interface provides no implementation, only method specifications and optional class-scope attributes
  - All methods are implicitly *abstract*, meaning they have no bodies
- Example:
  - Order provides an API "contract" between clients and implementations



REALIZATION DISCUSSION

Really: *implements* (realize) vs *extends* (generalize).

- Realization provides the significant benefits of polymorphism without the problems of generalization
  - Provide ability to *decouple* parts of a system without added *tight coupling* inherent to generalization
  - Design leverages interfaces heavily and we will see many examples later in the course
  - Must modern object languages (Java, CORBA, C#) and technologies (patterns, frameworks) emphasize *interfaces*

Liskov Substitution Principle

If S is a subtype of T, then objects of type T in a program may be substituted with objects of type S, without altering that program.

That is, we can use a child in place of a parent.

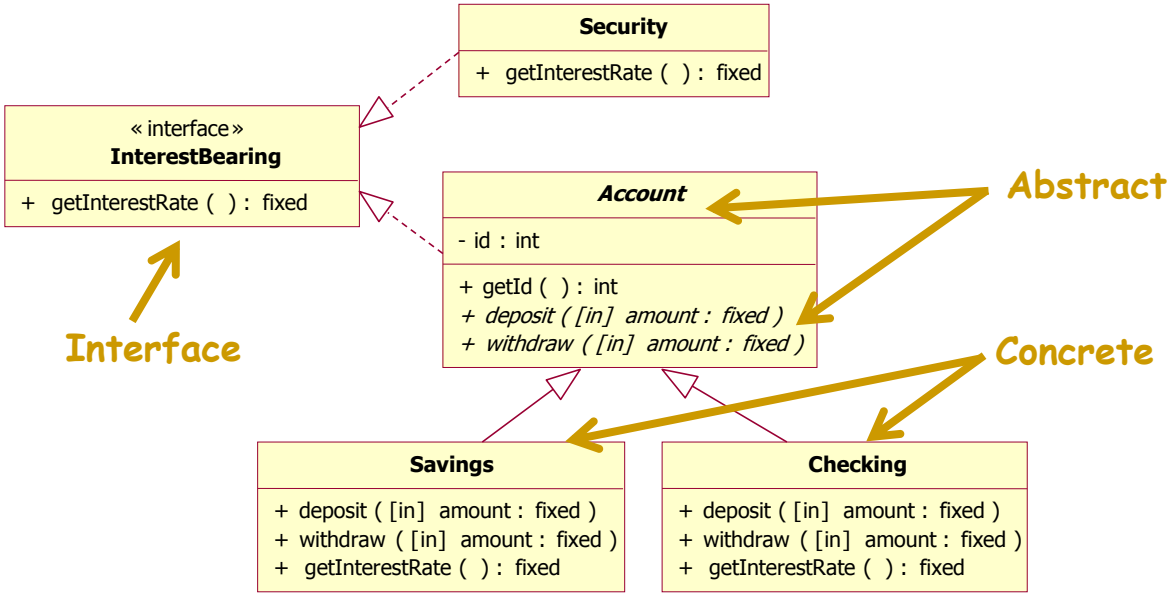
Be careful: what happens if we have a Square derive from Rectangle?

We'll end up with a Square class containing methods like getHeight or setWidth. Not good!

- Allows for easy, planned system extension
  - Modify system without modifying, or even recompiling, clients
- Supported by both generalization and realization relationships
  - Realizations provides substitution without introducing the tight coupling of generalization

POLYMORPHIC CLASS SUMMARY

Interface	*no implementation*, all methods abstract
Abstract class	*partial implementation*, some methods are abstract (italicized in UML, or can use {abstract}) while some may have an implementation
Concrete class	*all methods implemented*; can create instances



WHAT IS A COMPONENT?

We've talked plenty about Class Diagrams, let's change gears to talk about Component Diagrams. Component Diagrams express the high level structure of our system in terms of "components".

- It's not that easy.
  - There are many definitions of component because there are many perspectives
    - Developer – decouple behavior into components
    - Configuration manager – package behavior into deployable components
    - Deployer – understand & manage versions of and dependencies between components
- But, there are some agreed upon characteristics …

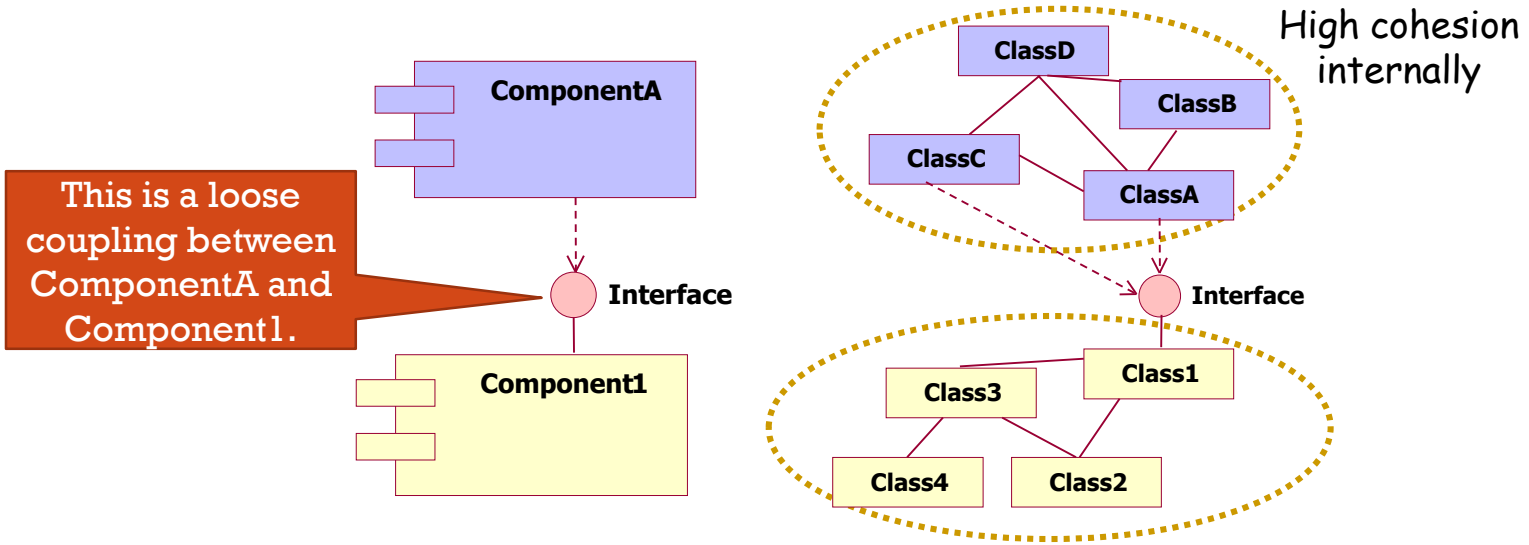
"A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces…typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations." – UML User's Guide

"A unit of composition with contextually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition." – Clemens Szyperski

"A nontrival, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture." – Philippe Krutchen

LOW COUPLING, HIGH COHESION

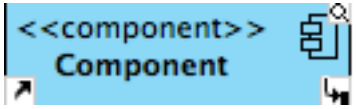
- Components strive to provide highly cohesive functionality in a manner that facilitates decoupled systems



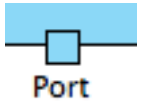
- This is essentially the Liskov Substitution Principle discussed before applied to components
  - Component provides a *realization* of a coarse business function
  - New components (better QoS – Quality of Service) may replace old ones easily (*decoupled*)
  - Implementation is highly *cohesive* to optimize the component behavior

UML COMPONENT DIAGRAM SYNTAX

- From Visual-Paradigm:

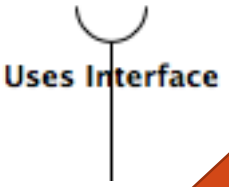


- Component (may be nested)



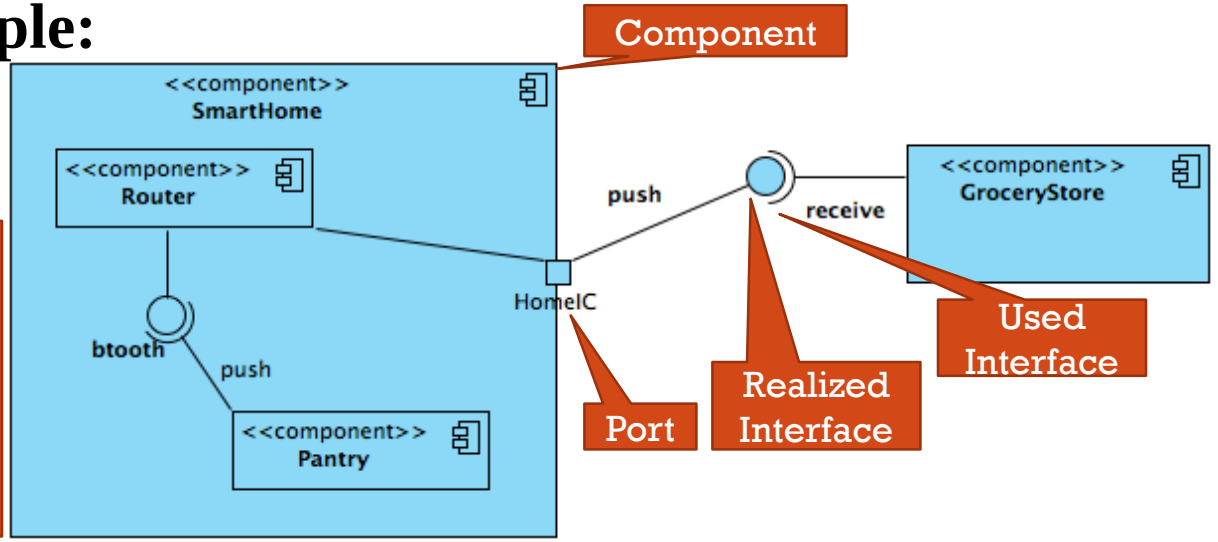
- Port: Exists on a component to indicate I/O

- Interface implemented by the component



- Interface the component depends on (uses)

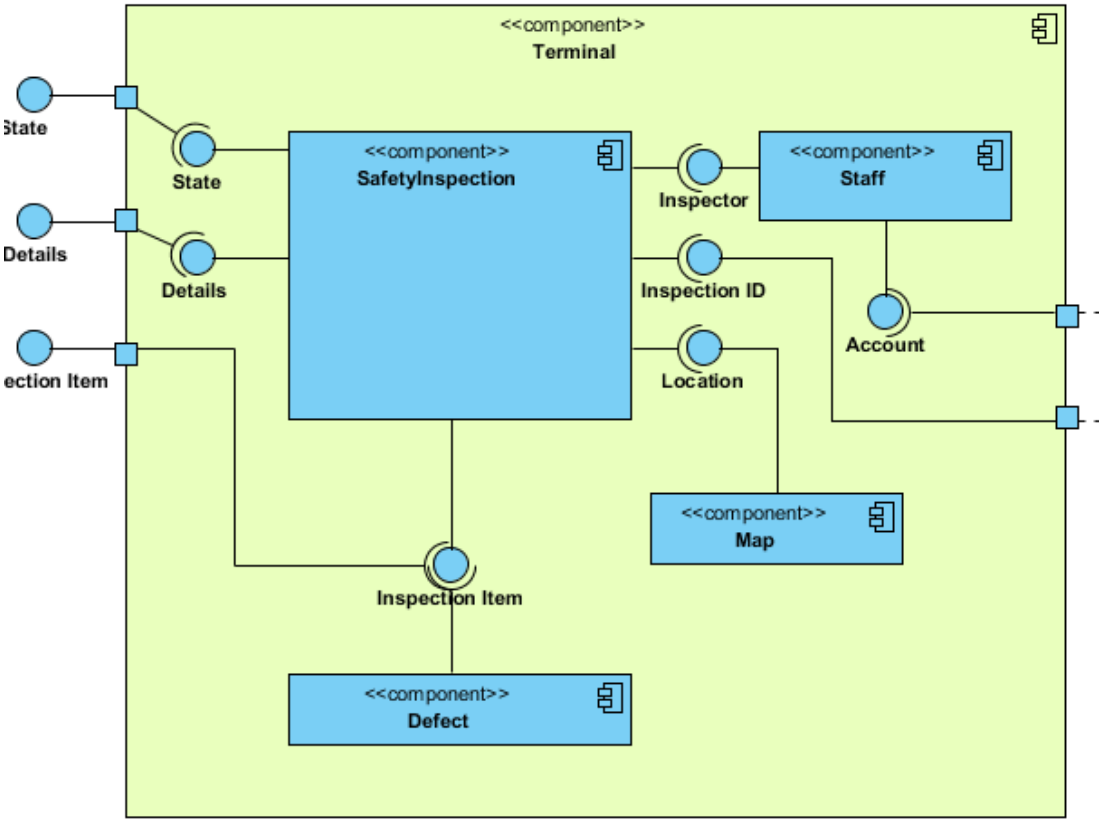
Example:



Pantry is a component that uses the *btooth* interface, which is realized by the Router component. The Router component uses the interface *push* to communicate with the GroceryStore component outside of the SmartHome nested component.

EXAMPLE: UML COMPONENTS

Try reading this on your own. In general, component diagrams aren't too hard to understand since they use a limited variety of elements.



Also see: http://www.uml-diagrams.org/component-diagrams.html
This site has more explanation on Component Diagrams, a little detailed though.

UML ELEMENTS AND STRUCTURAL DESIGN WRAP-UP

- Again, UML is a ***language*** so it has rules and semantics
- Structural design strives to place system responsibilities in the "best" location to support the system's objectives
  - Typically strive for decoupling and separating concerns
  - Objectives can be conflicting so there may be no best solution
  - Many design-level decisions are well documented with *patterns*
- Class diagrams abstract from the instance world
  - When modeling (OOA&D), can be used to capture templates of common object conversations
  - Bottom-up in this sense (note this is *not* an OOP view!)
- Components *realize* coarse-grained system behavior
  - Top-down OOA&D starts by identifying components or subsystems
  - Give us a bigger "bucket" in which to assign responsibilities
  - Then we can determine what goes in the component implementation

A NOTE ON PROCESS

- For the 1st half of the semester, we focused on UCD process
  - We also mentioned Software Engineering Processes, and Engineering Design Process.
- What about processes for creating our UML designs?
  - There are a number of methodologies out there, most of which focus on "islands" of work or a mishmash of techniques.
  - We suggest moving to EDP – assuming your context and requirements are mature, you should be able to move away from user feedback.
  - You may also choose to continue with UCD, although, it will not be beneficial.
    - Perhaps it is a good idea to continue UCD for your UI design, to refine your use cases, etc., as needed for the UML design.