Advanced Modeling

Part I

Advanced Classes

Advanced Classes

- Classes are the most important building blocks of an object-oriented system
- Classes are not only that: in UML, many things are represented as classes (classifiers)
- A classifier is a general mechanism that describes structural and behavioral features
- Classes, actors, interfaces, data types, signals, components, nodes, use cases and subsystems are classifiers
- They are all subject to the same rules as basic “classes”

Kinds of Classifiers

- **Interface**: A collection of operations that are used to specify the services provided by a class
- **Datatype**: As in C++, classes can be used to represent data types (value and operations)
- **Signal**: The specification of an asynchronous stimulus communicated between instances of classes
- **Component**: A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces

Kinds of Classifiers

- **Node**: A physical entity that exists at run time and that represents a computational resource, generally having at least some memory and processing capabilities
- **Use case**: A description of a set of sequences of actions, including variants, that a system performs that yields an observable result of value to a particular actor
- **Subsystem**: A grouping of elements of which some constitute a specification or behavior offered by the other contained elements

Additional Properties of Classes

- **Visibility**: The visibility of a classifier feature specifies whether it can be used by other classifiers (as in C++)
  - public (+) , protected (#) and private (-)
- **Scope**: The scope of a classifier feature specifies whether the feature appears in each instance of the classifier, or whether there is just instance for all the instances of the classifier (static attributes and operations in C++)
  - instance (default), classifier (underlined feature)
- **Abstract elements**: Abstract classes and features are elements that find their instances in the child classes in the hierarchy. They are represented by the italics font.

Additional Properties of Classes

- **Root elements**: A root classifier is a class that cannot have a super-class. Represented using the {root} tag.
- **Leaf elements**: A leaf classifier is a class that cannot have a child class. A leaf feature is a feature that is non-polymorphic. It cannot be overriden by a feature in a child class. Represented using the {leaf} tag.
- **Multiplicity**: The number of instances a classifier or feature can have at run-time.

Additional Properties of Classes

- **Attributes**: The type, initial value and changeability of each attribute can be specified
- General syntax:
  - **[visibility] name [multiplicity] [: type] [= initialValue] [{property}]**
    - origin
    - + origin
    - origin : Point
    - name [0..1] : String
    - origin : Point = (0,0)
    - pi : Integer = 3.1416 {frozen}
- changeable: no restrictions on modification (default)
- addOnly: additional values may be added, but not removed
- frozen: value may not be changed after initialization

Additional Properties of Classes

- **Operations**: The return type, parameters' types, concurrency semantics, etc. can be specified
- General syntax:
  - **[visibility] name([paramList]) [: returnType] [{property}]**
  - display
  - + display
  - set (n : Name, s : String)
  - getID() : Integer
  - restart() {guarded}
- General syntax for parameter specification:
  - **[direction] name : type [= defaultValue]**
  - direction may be **in, out** or **inout**

Additional Properties of Classes

- Additional properties of operations:
  - isQuery: Operation does not change the state of the instance
  - sequential: a function that should not be called concurrently
  - guarded: A function that cannot be called concurrently
  - concurrent: A function that is designed to cope with multiple concurrent flows of control
  - The last three ones are relevant only in the presence of active objects, processes and threads.

Part II

Advanced Relationships

Advanced Relationships

- **Dependencies**: The most general relationship available. Semantically speaking, all relationships are dependencies, but with their own flavor. Dependencies are used to show that one thing is using another.
- Because of this generality, there are 17 different stereotypes that can be used to give a shade of meaning on relationships.
- The most common use of straight dependencies is to show that a class uses another because one of its operations uses it as a parameter.
- Rendered as a dashed directed line with an open arrowhead

Dependency Stereotypes

- derive: specifies that the source can be computed from the target, e.g. age can be derived from birth date.
- friend: specifies that the source is given special visibility into the target (same as friends in C++)
- instantiate: specifies that the source creates instances of the target
- refine: specifies that the source is of finer degree of abstraction than the target, e.g. relations between versions

Dependency Stereotypes

- access (packages): a package can refer to elements of another package, e.g. Class::attribute.
- import (packages): a package can freely use elements of another package
- extend (use cases): specifies that a use case extends the behavior of another.
- include (use cases): specifies that the source use case incorporates the target use case, e.g. decomposition of use cases as reusable parts

Advanced Relationships

- **Generalizations**: A relationship between a general thing (the superclass) and a more specific thing (the subclass).
- The subclass inherits all features of the superclass, overriding multiply defined features.
- Multiple inheritance is possible but must be used carefully
- Can be used on any classifier (including classes, but also on use cases, actors, interfaces, etc.)
- Rendered as a solid line with a closed arrowhead

Generalization Stereotypes

- complete: specifies that all children in the hierarchy have been specified. No more children can be created.
- incomplete: specifies that some more children can be created. Normally denotes an unfinished hierarchy.
- disjoint (multiple inheritance): specifies that objects of the source may have no more than one of its parents as a type (dynamic type allocation not allowed).
- overlapping (multiple inheritance): specifies that objects of the source may have more than one of its parents as a type (dynamic type allocation allowed).

Advanced Relationships

- **Associations**: A structural relationship specifying connection (navigability) between objects of two classes. This is the "has a" relationship used in entity-relationship diagrams.
- Association *name, roles* and *multiplicity* are often defined
- Rendered as a solid, normally undirected line.
- Aggregation: a special case of association that a "part of" relation between source and target. Represented as a solid line with a diamond on the target side.
- Composition: a special case of aggregation stating that the target's lifetime is linked to the source's lifetime. Represented as a solid line with a solid diamond on the target side.

Advanced Associations

- visibility specifiers (roles): specifies the visibility of elements in a relationship, similarly to the visibility specifications of attributes and operations (+, #, -) (figure 10-4)
- interface specifiers (roles) : specifies that a role is using a specific interface realized by the target for communication. Useful when the same class can have different roles which use different interfaces. (figure 10-6)

Advanced Relationships

- **Realizations**: A semantic relationship between classifiers in which the source specifies a contract that the target guaranties to carry out. The target is normally an abstraction of the source.
- Used most of the time to specify the relationships between interfaces and and its realizer class, or between use cases and its realizer collaboration.
- Rendered as a dashed directed line with a closed arrowhead.