A Schema for UML Notation (Review)

UML Building Blocks
- **Things**: The basic entities in the model.
- **Relationships**: Tie things together.
- **Diagram**: Is a graph of things and their relationships.

Building Blocks of UML

Four Kinds of “things”

Seven Kinds of Structural Constructs in UML

Structural Thing: Class
- A common description of a set of objects.
Structural Thing: Interface
• A collection of operations provided by a class of a component.

Structural Thing: Collaboration
• A collection of structural elements and behavioral elements.

Structural Thing: Use case
• A sequence of actions. A structure superimposed over behavioral things.

Structural Things: Active Class
• A class whose objects can initiate a thread or a process.

Structural Things: Component
• A physical packaging of classes, interfaces, and collaborations.

Structural Things: Node
• A physical element with memory and processing power.
Four Kinds of “things”

- Structural things
- Behavioral things
- Grouping things
- Annotational things

Behavioral Building Blocks

- UML Building blocks
- Things
  - Structural things
  - Behavioral things
  - Grouping things
  - Annotational things
  - Interaction
  - State machine

Dynamics of the Model: Behavioral Things

- Interactions: The “verbs” of the model. A set of messages exchanged among a set of objects.
- State Machine: A sequence of states that an object goes through.

Organization of a model: Grouping Things

- Package: A general-purpose mechanism for bundling together structural, behavioral, or other packages.
Explanations: Annotational Things

- Notes: Explain the model, comments, constraints, etc.

This class is for ....

Building Blocks of UML

Relationships in UML

- Dependency: Change to one thing will affect the other.
- Association: Set of links between objects.

Relationships cont’d

- Realization: A specification of a contract between two entities.

Building Blocks of UML
Advanced Structural Notation in UML

Extensions to UML: Stereotypes

- **Class** is a basic modeling construct in UML. How do you add constructs like **Class** to the language?
- UML gives you a “meta-class” called **stereotype**.

- An instances of a **stereotype** give you a thing that is equivalent to a class.

### Meta-class, class, instances

<table>
<thead>
<tr>
<th>&lt;&lt;Meta-class&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>Data Members</td>
</tr>
<tr>
<td>Function Members</td>
</tr>
<tr>
<td>UML Symbol</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Class</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Customer</strong></td>
</tr>
<tr>
<td>char* CustName;</td>
</tr>
<tr>
<td>int CustId;</td>
</tr>
<tr>
<td>print( );</td>
</tr>
<tr>
<td>display( );</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Instance</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe: Customer</td>
</tr>
</tbody>
</table>

### Stereotypes

<table>
<thead>
<tr>
<th>&lt;&lt;Stereotype&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>Data Members</td>
</tr>
<tr>
<td>Function Members</td>
</tr>
<tr>
<td>UML Symbol</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;&lt;exception&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>int UflowAmount;</td>
</tr>
<tr>
<td>int LineNumber;</td>
</tr>
<tr>
<td>Throw( );</td>
</tr>
<tr>
<td>Catch();</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Instance</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>uf: Underflow</td>
</tr>
<tr>
<td>UflowAmount=5</td>
</tr>
<tr>
<td>LineNumber=51</td>
</tr>
</tbody>
</table>
Icons for Stereotypes

Extensions

- **Classes** have properties: name, set of attributes, operations, etc.
- **Associations** have properties: name, end points, etc.
- **Stereotypes** add new “things” to the language.

How do you add new properties to things?

Extensions: Tagged Values

- **Tagged Values** add properties to things!
- Tagged values are a piece of “Meta-data”, i.e. not the same as attributes or data members.
- Can be added to existing elements or stereotypes.

Symbol for Tagged Values

Uses of Tagged Values

Tagged values are useful for adding properties about:

- the code generation,
- version control,
- configuration management,
- authorship,
- class to language mapping, etc.
Extensions: Changing semantics

- Each thing in UML has an associated semantics, e.g., **Generalization** implies one-to-one substitution:
  
  ![Diagram of Generalization](image)

  \[
  \text{Map}(\text{instB}, \text{instA}) \text{ implies: for each member } X \text{ of Class}_A \\
  \text{instA}.X = \text{Map}(\text{instB}.X, \text{instA}.X);
  \]

Semantic Modification: Constraints

- **Constraints**
  - UML has an associated Object Constraint Language (OCL) for formally specifying constraints.
  - Constraints can be specified as {pure text} or using OCL.
  - Automatic verification of constraints, their validity, and mutual compatibility are not always possible. Computationally, they lead to NP-Complete, NP-Hard, and intractable problems.

- **Classifiers**
  - **Classes** let us write abstractions with attributes and operations.
  - **Classifiers** are more general than classes.
  - **Classifiers** allow us to specify structural and behavioral features.
  - Modeling beyond attributes and operations, e.g., visibility, multiplicity, polymorphism, signatures, …
  - Include: class, interface, datatype, signal, component, node, use case, and subsystem.

Classifier

- **Fundamental difference**: there are things that can be instantiated (e.g. class) while other cannot be (e.g. generalization, package).
- **Definition**: Entities that can have instances are called classifiers. They have structural features in the form of attributes, and behavioral features in the form of operations.

Specialization of a Classifier

- **Classifier**
  - Class
  - Interface
  - Datatype
  - Signal
  - Diagram
  - Component
  - Node
  - Use case
  - Subsystem
Classifier: Class
• A common description of a set of objects.

Classifier: Active Class
• A class whose objects can initiate a thread or a process.

Classifier: Interface
• A collection of operations provided by a class of a component.

Classifier: Use case
• A sequence actions. A structure superimposed on top of behavioral things.

Classifier: Component
• A physical packaging of classes, interfaces, and collaborations.

Classifier: Node
• A physical element with memory and processing power.
Visibility

- Classifiers allow the visibility of each attribute and operator to be specified.
- Public = +
- Protected = #
- Private = -
- Used for selective data hiding.
- Same as C++.

Scope of Attributes

- For all attributes, we need to specify if it is one/class or one/instance.
- Scope can be instance or classifier (= static of C++).
- instance by default. Underline if you want the scope to be classifier.

Abstract Class

- Represent pure abstractions. They cannot have instances.
- Indicated by italicizing the name.
- You can have abstract operations (Virtual).

Leaf Class

- Constrained to have no sub-classes.
- Indicated by [leaf] under the name.

Multiplicity of a Class

- Number of instances a class may have.
- 0, 1, N, [1..5], *. 

Underflow

| + UflowAmount: int;  
| + LineNumber: int;  

# Throw( );  
# Catch( );

Customer
char* CustName;
int CustId;
char* species;
print( );
display( );

PersistentObject
char* ObjDesc;
int ObjId;
Save( );
Load( );

Customer
char* CustName;
int CustId;
print( );
display( );

OKbox
(leaf)

DialogBox
fields
values
display( );
ask( );

DialogBox

OKbox

DialogBox

value
values
display( )
ask( );
Attributes

- Have a name, visibility, scope, multiplicity, type, initial value.
- Syntax: [visibility] name [multiplicity] [:type] [=init_val] [{property_string}]
  - e.g. + origin [1] :point = (0, 0) {frozen}
  - {property_string} =
    - changeable
    - addOnly => if multiplicity > 1, than subsequent ones are add only.
    - frozen => not changeable (i.e. const in C++)

Operations

- Operation vs. Method: Operation is service that can be requested from an instance. Method is the implementation. Thus the same operation may have multiple methods in a hierarchy. The binding is done at the run-time.
- signature = {name, parameters, return valu}

Operation Syntax

[visibility] name [(parameter-list)] [: return-type] [{property-string}]

- Examples:
  + print
  setAge(age: Integer) : Integer
  schedule() {sequential}

- parameter-list has:
  [in/out/inout] name : type [def-value]

Operation Concurrency Control

[visibility] name [(parameter-list)] [: return-type] [{property-string}]

property-string:

- {leaf} => unrefinable
- {isQuery} => leaves the state unchanged
- {sequential} => for object integrity, make sure only one control flow thru the object is allowed.
- {guarded} => takes care of sequentialization
- {concurrent} => integrity guaranteed during parallel processing.

Operations

- In general, polymorphic.
- can have the same signature as a parent.
- Unrefinable operations indicated by {leaf} under the name.
- *Italic* implies an abstract operation. Must be defined by a sub-class

Template Classes

- A parameterized element.
- Has slots for classes, instances and values which serve as parameters.
- Must be instantiated before use. Instantiation leads to an ordinary class.
Example: Lookup Table

Definition:
template<class Item, int maxSize>
class LookUp {
  public:
    virtual insert(const Item&);
    virtual Boolean query(const Item&)
  ...
};

Usage:
LookUp< Person, 1000> personLookUpTab;

Summary

- There is a rich set of modeling premitives.
- The language is extensible.
- We have covered all the “structural things.”
- Next class:
  - Relationships
  - Diagrams

UML Template Notation

```
Item
MaxNum: integer

LookUp
+ insert(in I: Item): Boolean
+ query(in I:Item) : Boolean [IsQuery]

<<bind>> (person, 1000)
explicit binding
PersonLookUpTab
```