Object-Oriented Programming

UML Class Diagrams

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

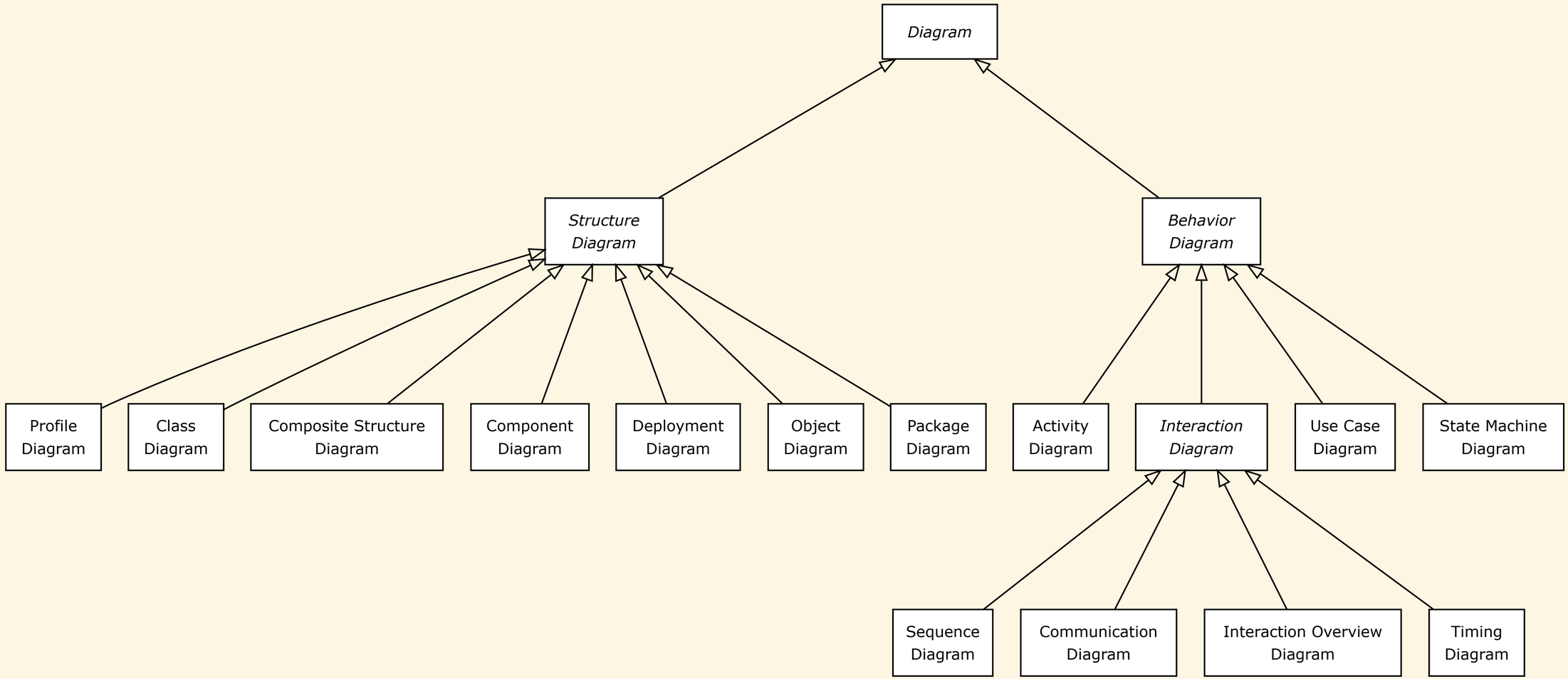
Communicating Class Design

- Textual description

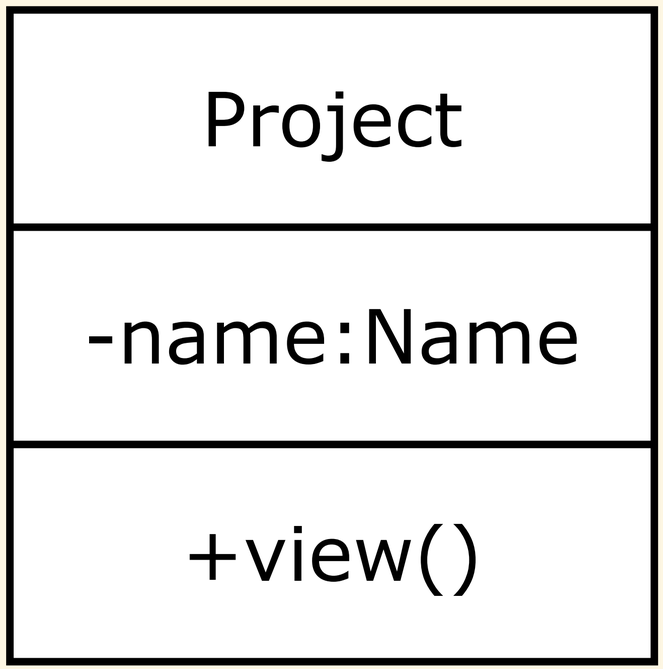
- C++ code

- UML Class

UML Diagrams



UML Class



- A static, structure diagram
- *Name*
- *Attributes* - data members/fields
- *Operations* - methods/member functions

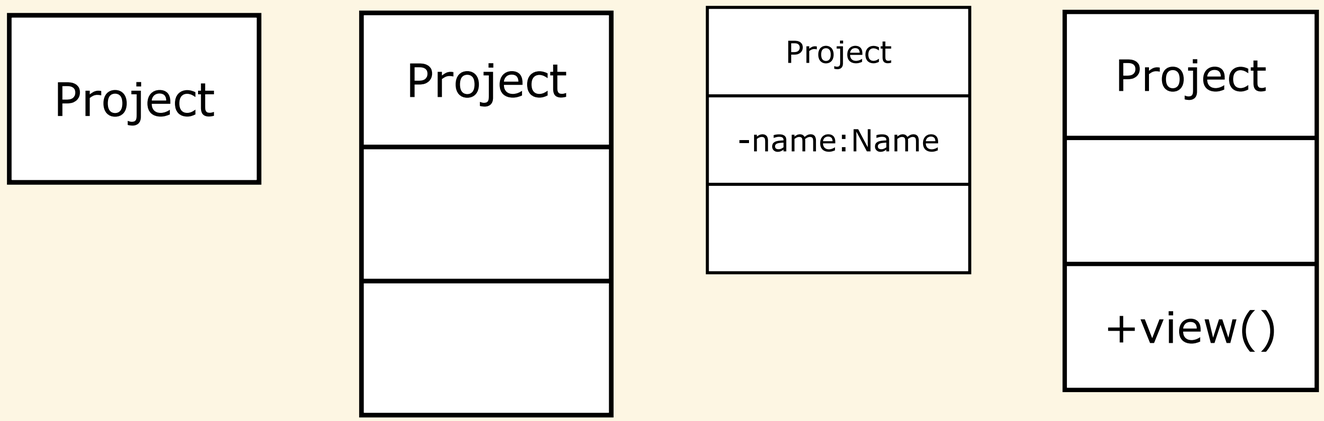
Partial *Meta-Object Facility* (MOF)

Level	Name	Description	Example
M2	*metamodel*	UML Syntax	Cat
M1	*model*	UML Model	Cat
M0	*data layer*	Real World	

Role of Classes

- Classes represent the *parts* of the program
- … they are *entities*, not *actions*
- … PascalCase, e.g., *Project*, *UndergraduateStudent*
- Operations are the actions of the class
- … like free functions, they are in the *verbTarget* form
- … typically use (at the model level) camelCase

Multiple Views



Visibility

- + public
- – private
- # protected
- ~ package

UML Primitive Types

- Integer
- UnlimitedNatural
- Real
- Boolean
- String

UML Primitive Types && C++

UML Type	C++ Type	C++ Standard Alternatives	Compiler Extensions (MSVC Italicized)
Integer	`int` (32)	`int8_t` (8), `int16_t` (16), `int32_t` (32), `int64_t` (64), `short` (16), `long` (32/64), `long long` (64)	`__int128` (128), *`__int8`* (8), *`__int16`* (16), *`__int32`* (32), *`__int64`* (64) (MSVC)
UnlimitedNatural	`unsigned int` (32)	`size_t` (32/64), `uint8_t` (8), `uint16_t` (16), `uint32_t` (32), `uint64_t` (64), `unsigned short` (16), `unsigned long` (32/64), `unsigned long long` (64)	`unsigned __int128` (128), *`unsigned __int8`* (8), *`unsigned __int16`* (16), *`unsigned __int32`* (32), *`unsigned __int64`* (64)
Real	`double` (64)	`long double` (80/128), `float` (32)	`__float128` (128), `__fp16` (16), `_Float16` (16), `_Float32` (32), `_Float64` (64), `_Float128` (128), `__bf16` (16)
Boolean	`bool`		
String	`std::string`	`std::string_view`, `std::u8string`, `std::wstring`, `std::u16string`, `std::u32string`, `const char*`, `char*`	

Attributes: *visibility name:type multiplicity = default {property-string}*

Attribute Syntax	Description
`+name`	public *name*
`-name`	private *name*
`-name:Name`	private *name* of type *Name*
`-name:Name = "Project"`	private *name* of type *Name* with default "Project"
`-name:Name[3]`	private *name* of type *Name* with multiplicity of 3
`-name:Name = "Project" { persistent }`	private *name* of type *Name* with the default "Project", and property 'persistent'

Operations: *visibility name(parameter-list):return-type {property-string}*

Operator Syntax	Description
`+draw()`	public operator *draw()*
`-draw()`	private operator *draw()*
`#draw()`	protected operator *draw()*
`+draw() : Boolean`	public operator *draw()* with return type Boolean
`+draw() : Boolean {optional}`	public operator *draw()* with return type Boolean, and a property *optional*

Operation Parameters: *direction name:type = default*

Parameter Syntax	Description
`+draw(: Shape)`	parameter of type *Shape*
`+draw(s: Shape)`	parameter *s* of type *Shape*
`+draw(in shape : Shape)`	in parameter *shape* of type *Shape*
`+draw(out picture : Picture)`	out parameter *picture* of type *Picture*
`+draw(inout picture : Picture)`	inout parameter *picture* of type *Picture*

Parameter Directions in C++

C++ Declaration	UML Parameter Direction
Shape	in
const Shape&	in
const Shape*	in
Shape&	inout
Shape&	out
Shape*	inout
Shape*	out
Shape**	out (typical API usage)

[Sutton'05]

Multiplicity

Multiplicity Syntax	Description
`*`	Any number of values
`1`	Single value (default)
`0..1`	No more than a single value
`2..4`	Two to four values

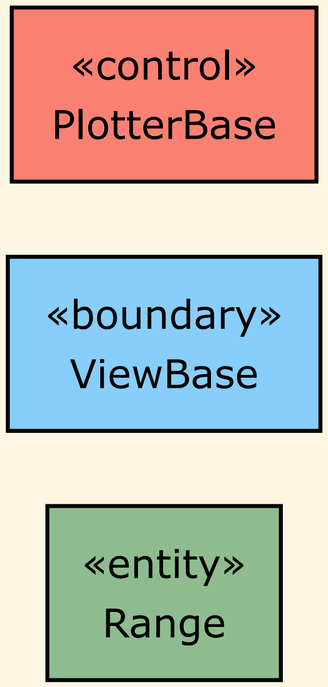
Multiplicity Terms

Multiplicity Term	Description
Optional	`*`
Mandatory	`1..*`
Single-valued	`0..1`
Multivalued	`0..2`, `0..*`

- For a multivalued UML attribute, use a container, e.g., in C++, `std::array`, `std::vector`, `std::deque`, `std::list`, `std::forward_list`

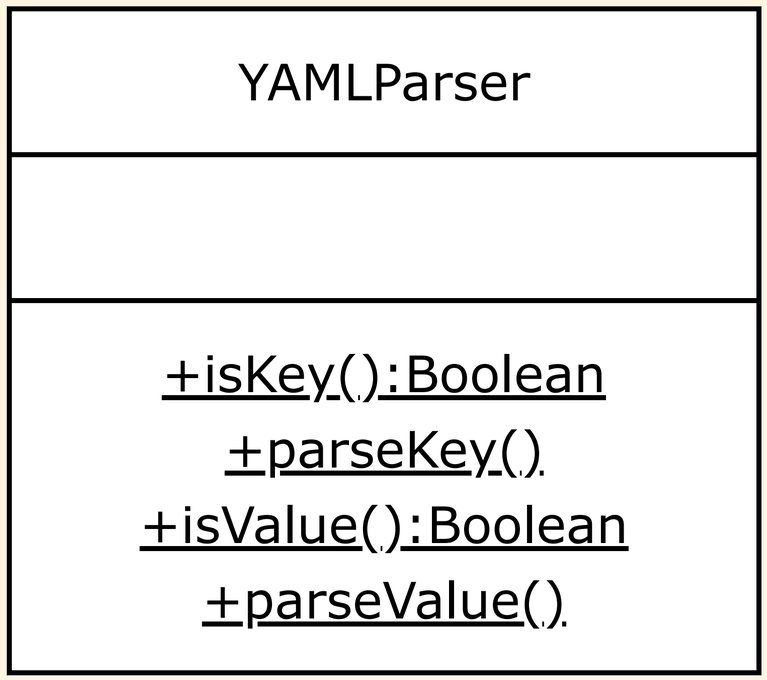
- For single-valued UML attribute, use an optional type or pointer if not available, e.g., in C++, `std::optional` (C++17)

Stereotypes



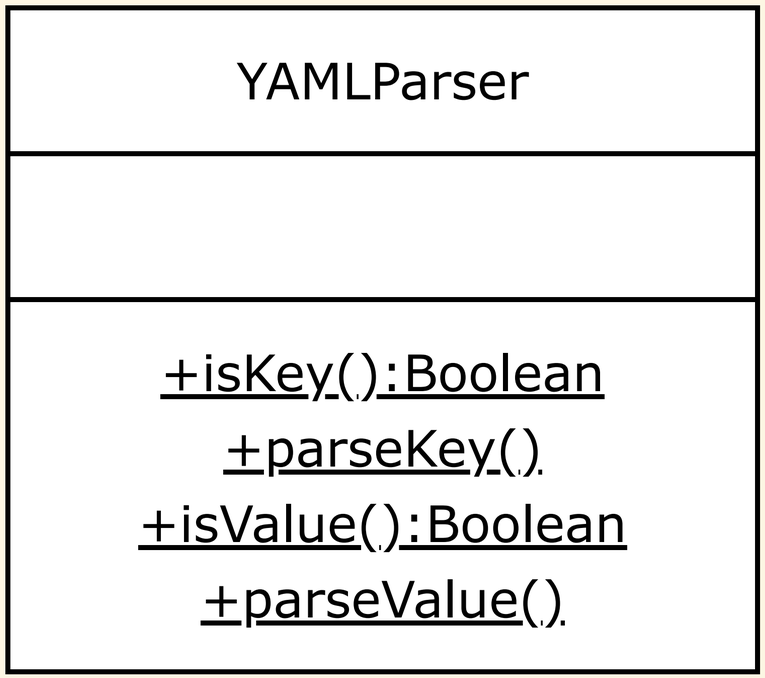
- *stereotype* is an extension mechanism to UML used as part of profiles
- *keyword* is a formally defined stereotype
- *CBE* - Control, Boundary, Entity

Static Attributes and Operations

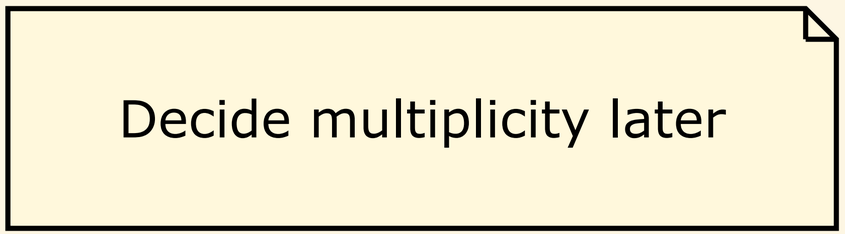


- No free functions in UML
- Static operations do not have an *object*
- Static attributes are shared among all objects of the class and can be used (in static operations) even without an object
- Operations and attributes that are static are indicated in UML with an underline
- Note: No parameters in UML does not mean there are no parameters. In this case, we don't care about that level of detail. The level of detail depends on the purpose of the model.

Static UML & C++



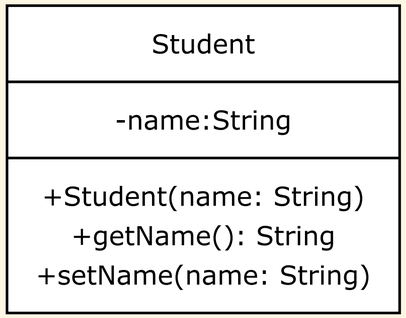

```cpp
class YAMLParser {
public:
    static bool isKey(/* ... */);
    static void parseKey(/* ... */);
    static bool isValue(/* ... */);
    static void parseValue(/* ... */);
};
```

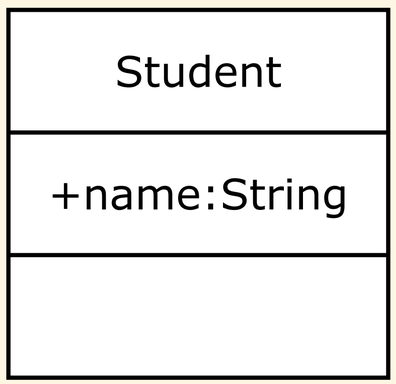


Notes

- Use text to clarify diagram/design
- The UML "comment" mechanism
- No formal definition of note content

Guidelines





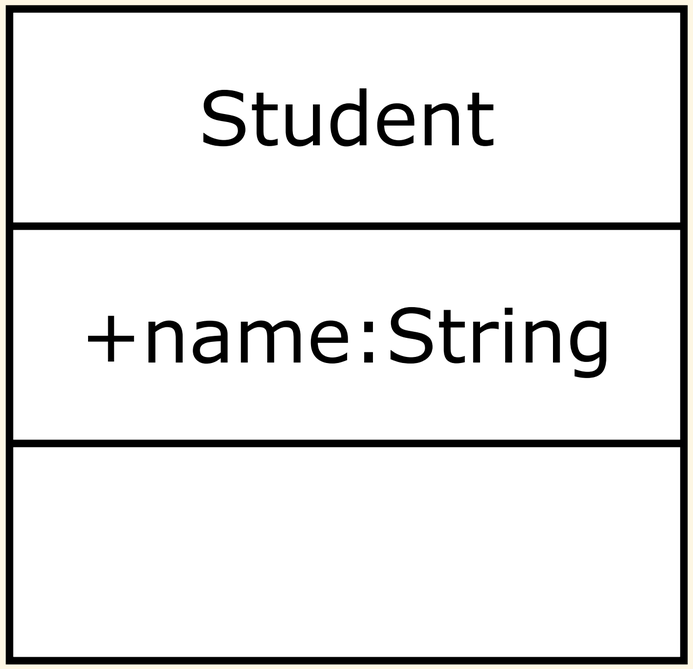
- Attributes show the state of the objects and may not map directly into types in the implementation language

- Show only the attributes and operations (methods) necessary for the given purpose

- Public attributes typically map to *get* and *set* (stereotype) methods in the code

- Only display constructors if needed

Forward Engineering: Model to Code




```
class Student {
public:
    // constructor
    Student(const Name& name);

    // name accessor
    const Name& getName() const;

    // name mutator
    void setName(const Name& name);

private:
    Name name;
};
```

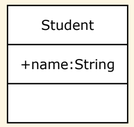
Reverse Engineering: Code to Model

```cpp
class Student {
public:
    // constructor
    Student(const Name& name);

    // name accessor
    const Name& getName() const;

    // name mutator
    void setName(const Name& name);

private:
    Name name;
};
```



UmlMode [Fowler]

- *UmlAsSketch*

- *UmlAsBlueprint*

- *UmlAsProgrammingLanguage*

UmlAsSketch

- UML to communicate an aspect of a system
- *forward engineering* - sketches to work out design before implementation
- *reverse engineering* - sketches for comprehension
- Main feature: Selectivity
- Requirements: Need to be performed quickly and collaboratively using a whiteboard or paper

UmlAsBlueprint

- Express design enough to hand off to another group

- *forward engineering* - Every detail needed to implement the design

- *reverse engineering* - Every detail of what design is in the code

- Separation of labor: High-end designers versus "coders"

- Main feature: Completeness

UmlAsProgrammingLanguage

- Provide enough detail to compile executable code from UML

- Promise: "UML is more productive than current programming languages"

- Reality: Graphical languages are not always better, e.g., flow chart versus pseudocode

- Reality: The "best" languages are not always the most successful

Current View

And, IMHO, the UML standard started going off the rails when some started morphing it to be a programming language with formal semantics. Visual representations are, by their very nature, abstractions, not precise representations.

— Grady Booch (@Grady_Booch) June 30, 2018