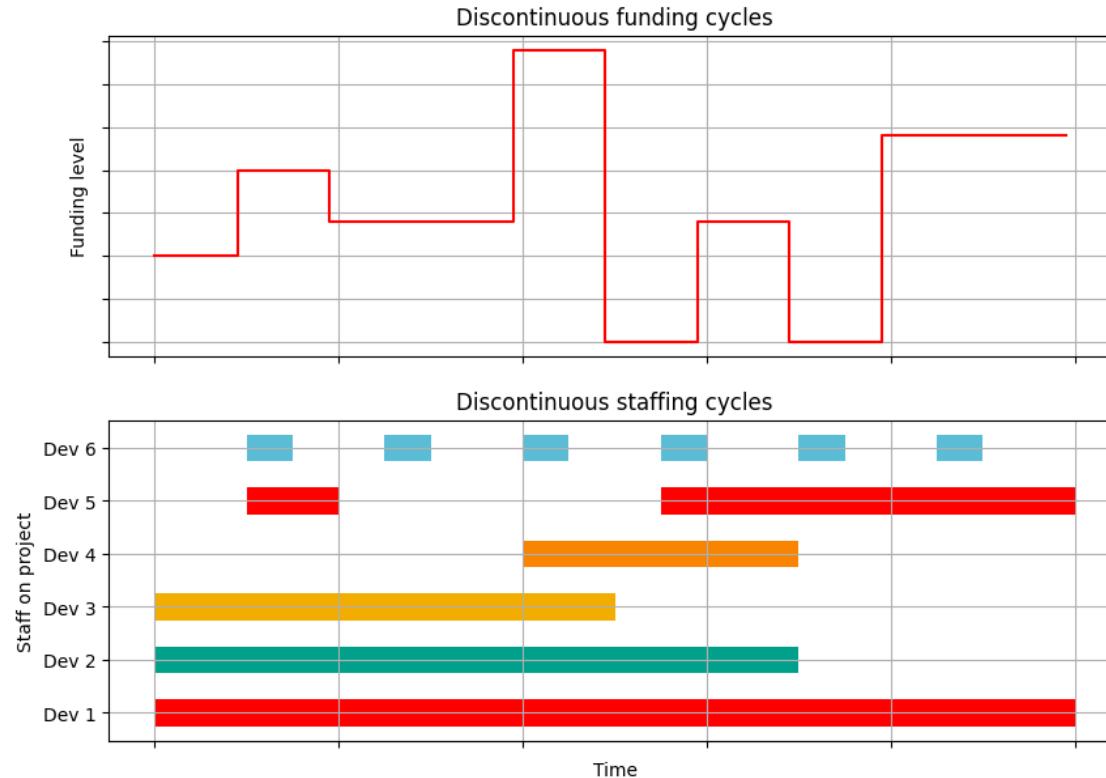


A nighttime satellite photograph of Earth, showing city lights as glowing yellow and orange dots against the dark blue and black oceans and continents.

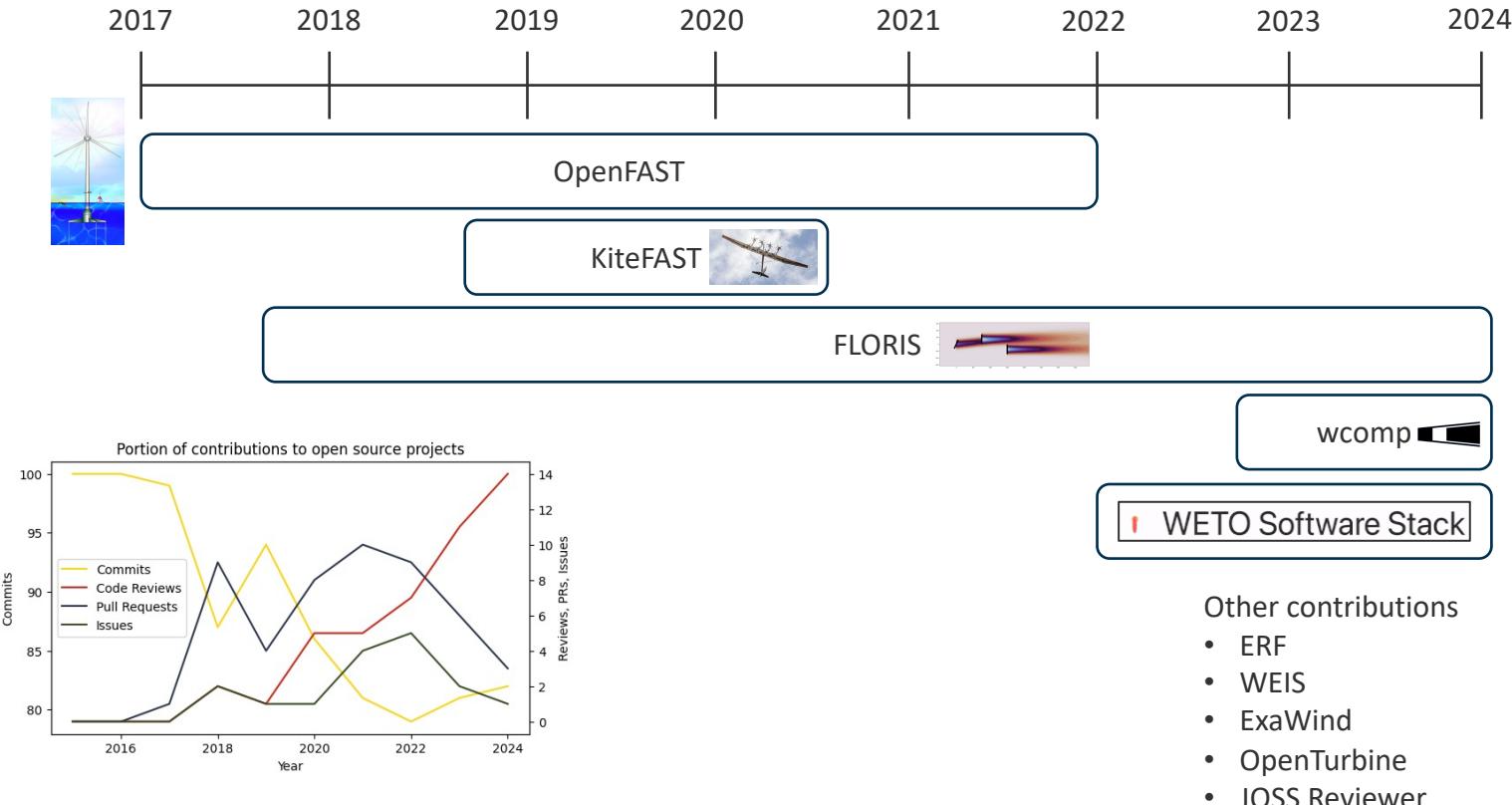
Strengthening development workflows  
by graphically communicating elements  
of software design

Rafael Mudafort  
IDEAS HPC Best Practices  
June 12, 2024

# Funding and staffing cycles

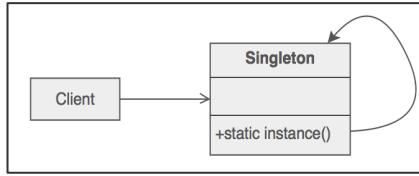


# About Me

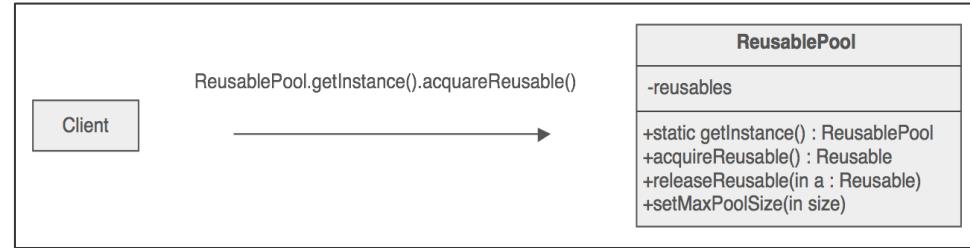


# Implicit design in software

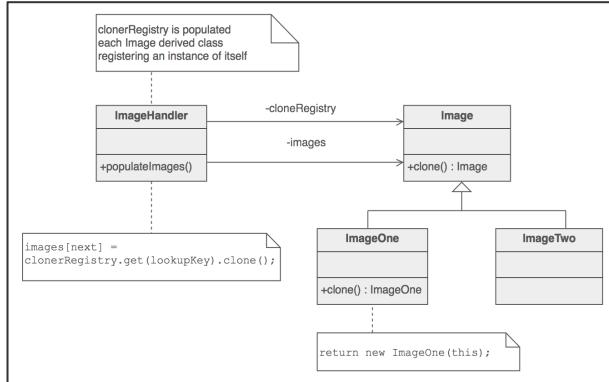
Singleton



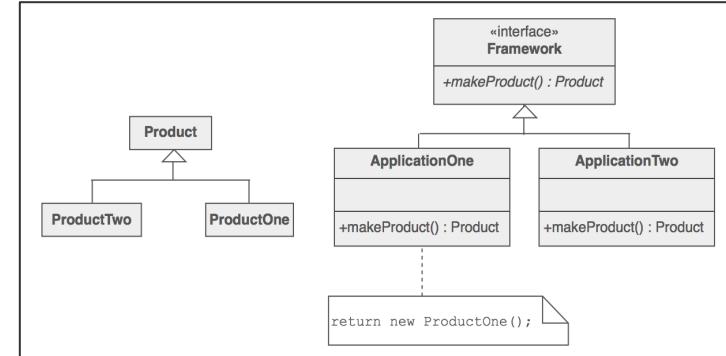
Object Pool



Prototype

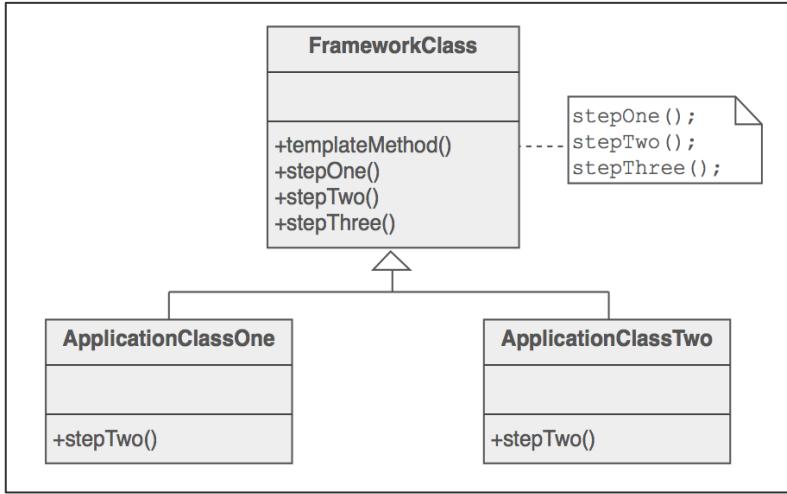


Factory

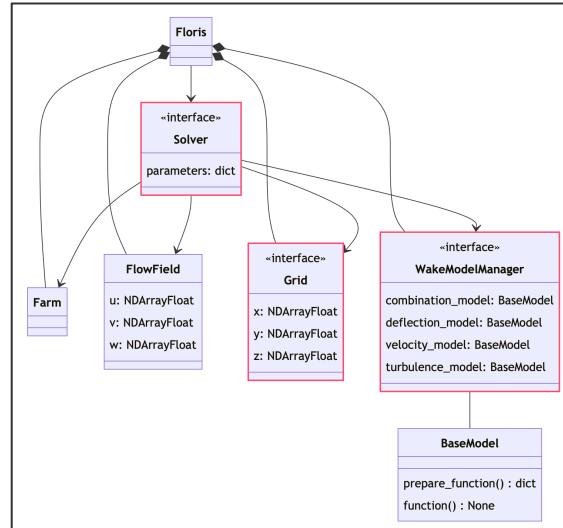


# Invest and scale

Abstract template design pattern



FLORIS template design pattern



1 : 1 → 1 : N

# Designed components of software

Component	Developer	User	Program Manager
System-level architecture	x	x	x
Package, module, subroutine architecture	x		
Design principles, overarching themes	x	x	x
Data structures and data flow	x	x	
Effective usage workflows		x	
Developer coordination and processes	x		

# UML: Unified Modeling Language

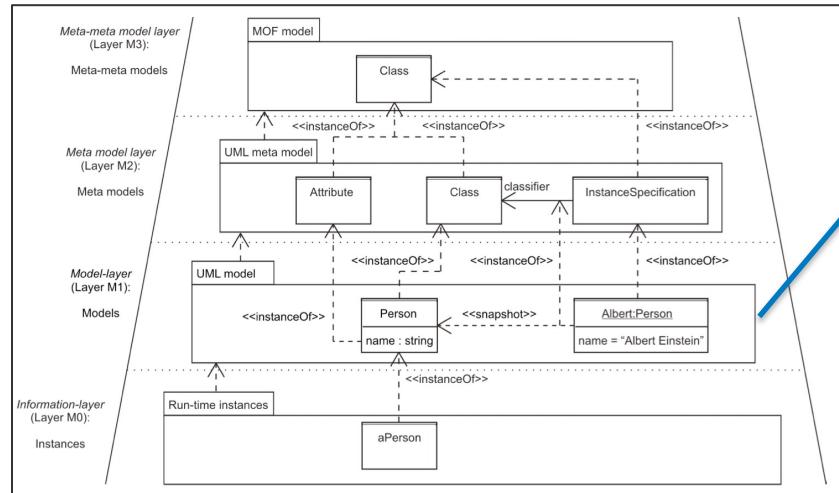
---

# UML: Unified Modeling Language

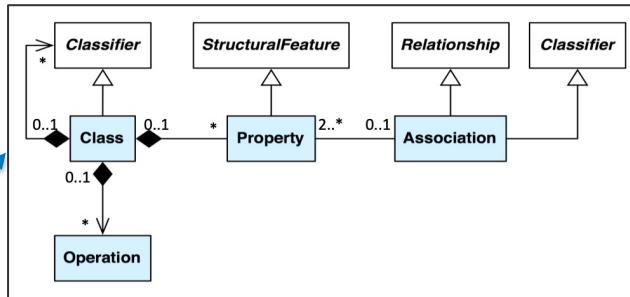


The Unified Modeling Language (UML) is a family of graphical notations, backed by single metamodel, that help in describing and designing software systems, particularly software systems built using the object-oriented (OO) style.

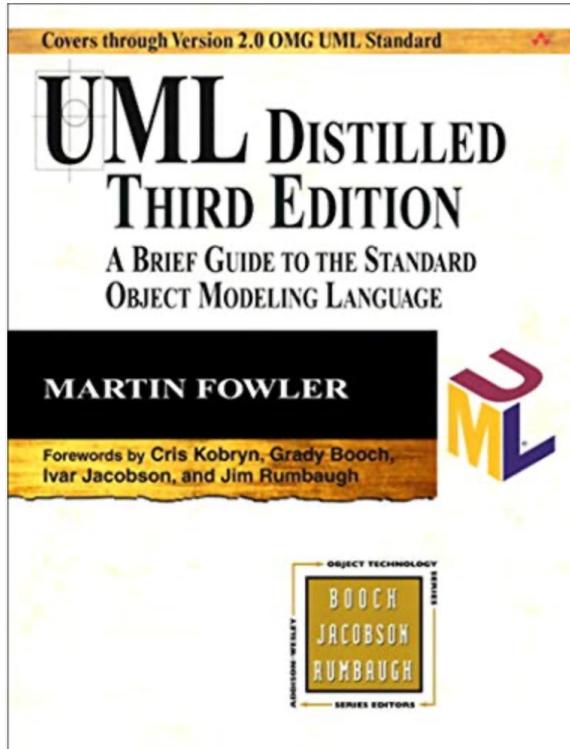
## Meta Object Facility (MOF)



## Class Diagram Model



# UML Distilled



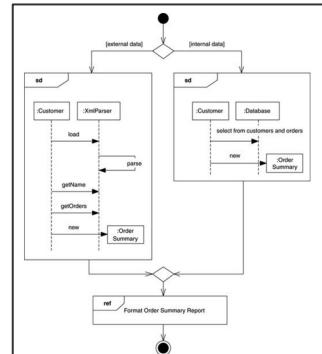
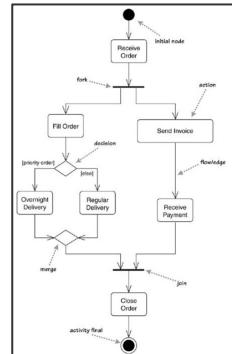
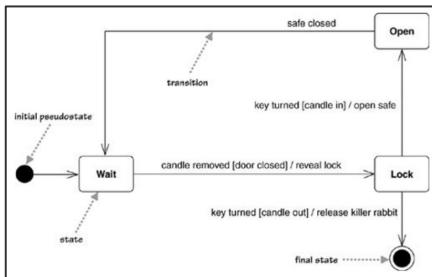
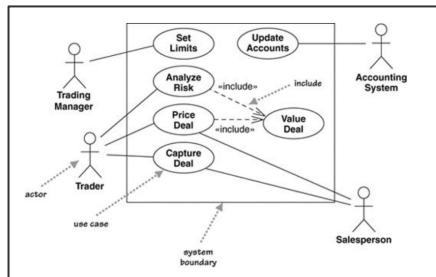
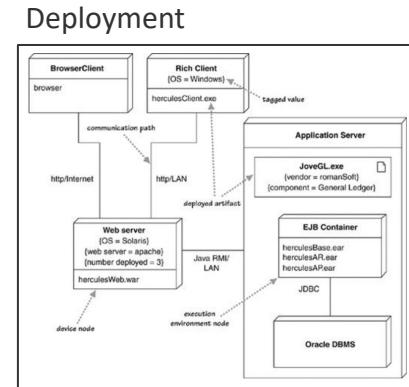
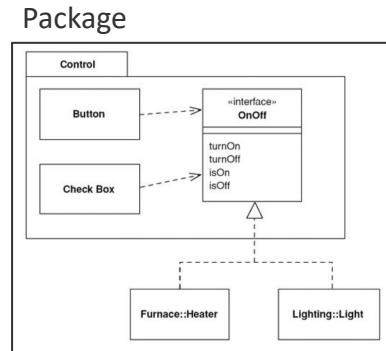
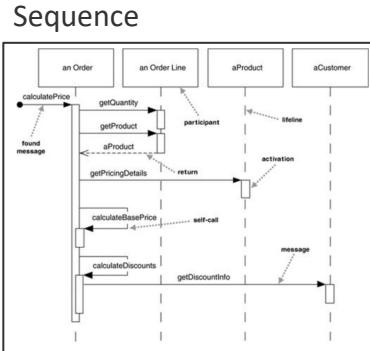
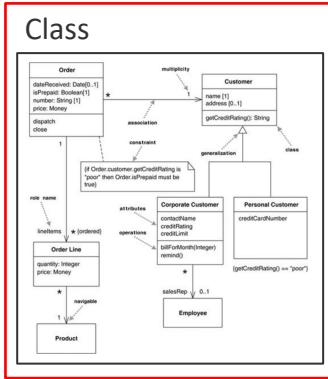
## Table of Contents

- Chapter 1: Introduction to UML
- Chapter 2: Development Process
- Chapter 3: Class Diagrams
- Chapter 4: Sequence Diagrams
- Chapter 5: Class Diagrams Advanced
- Chapter 6: Object Diagrams
- Chapter 7: Package Diagrams
- Chapter 8: Deployment Diagrams
- Chapter 9: Use Cases
- Chapter 10: State Machine Diagrams
- Chapter 11: Activity Diagrams
- Chapter 12: Communication Diagrams
- Chapter 13: Composite Structures
- Chapter 14: Component Diagrams
- Chapter 15: Collaborations
- Chapter 16: Interaction Overview Diagrams
- Chapter 17: Timing Diagrams
- Appendix: UML Versions

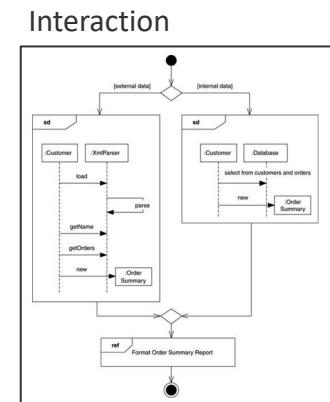
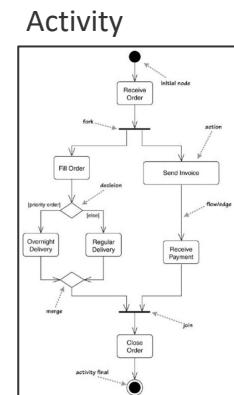
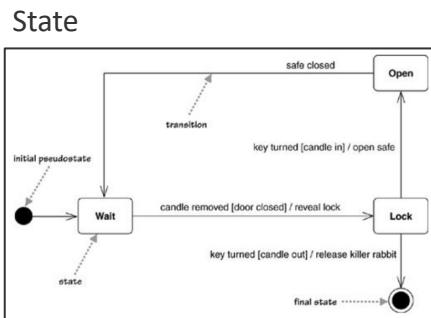
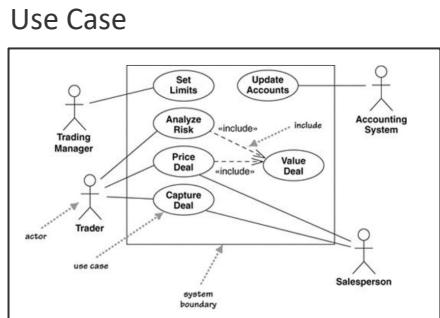
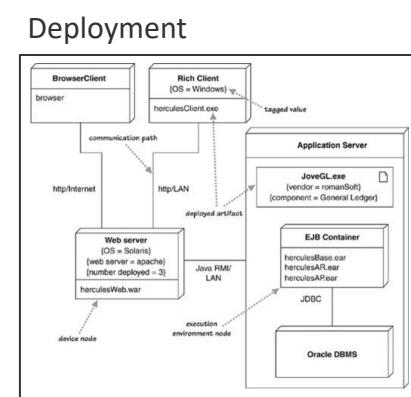
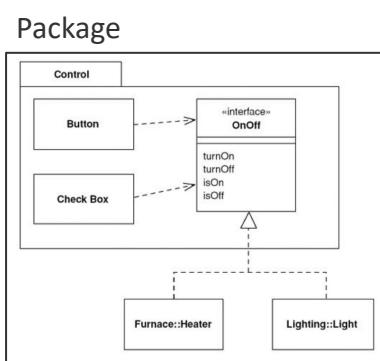
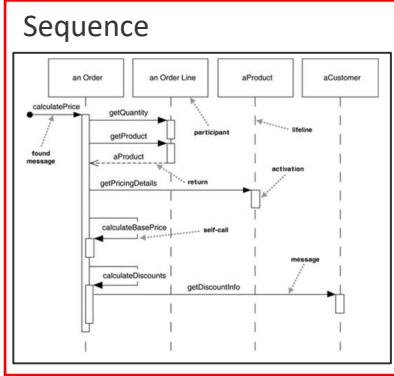
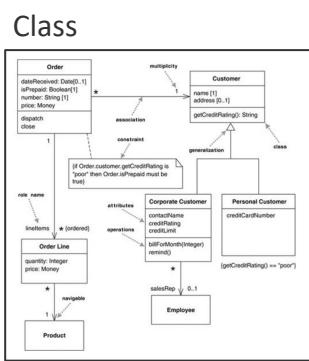
## Key features:

- *Development Processes* chapter
- *When to use X* for each diagram
- Short: 178 pages total
- Acts as a Reference and Explanation
- 1<sup>st</sup> Edition has a chapter on design

# UML: Unified Modeling Language

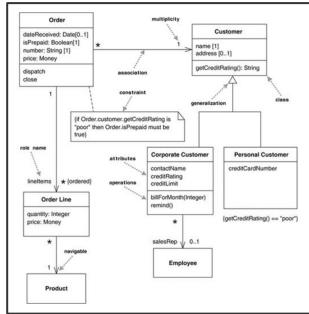


# UML: Unified Modeling Language

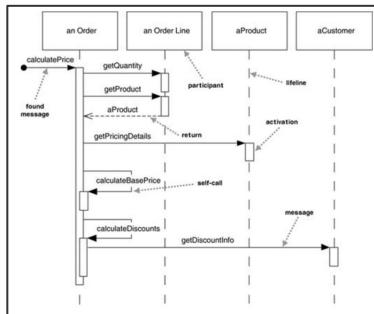


# UML: Unified Modeling Language

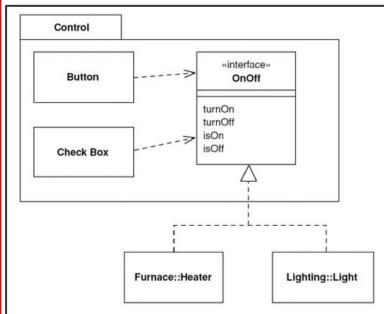
Class



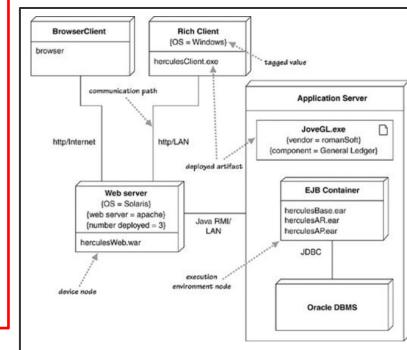
Sequence



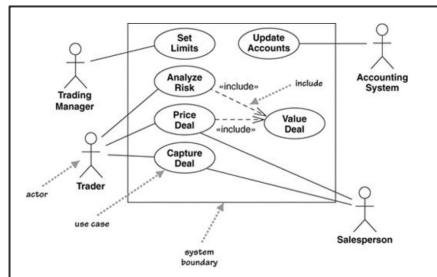
Package



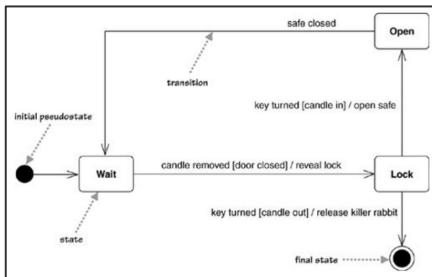
Deployment



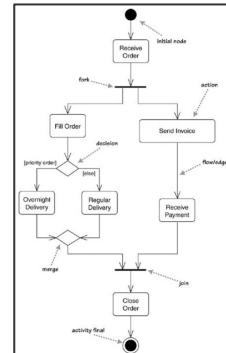
Use Case



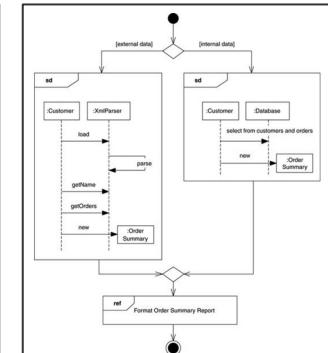
State



Activity

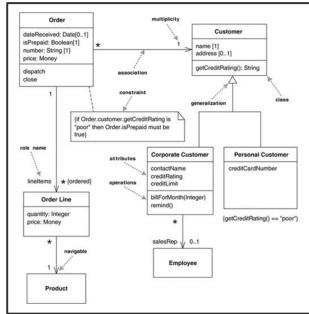


Interaction

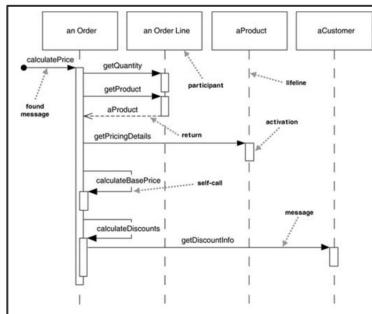


# UML: Unified Modeling Language

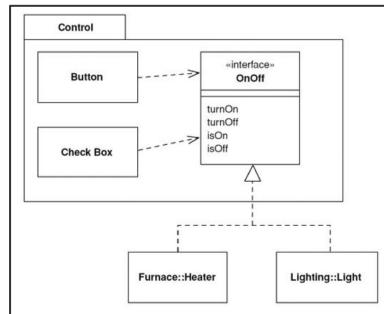
Class



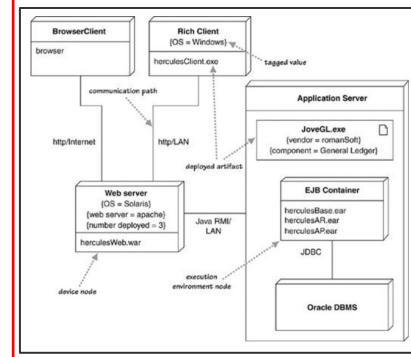
Sequence



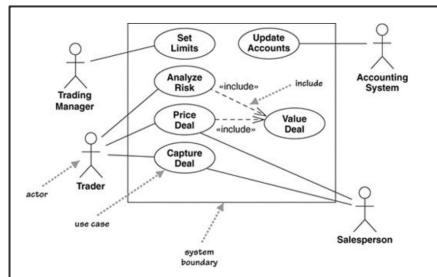
Package



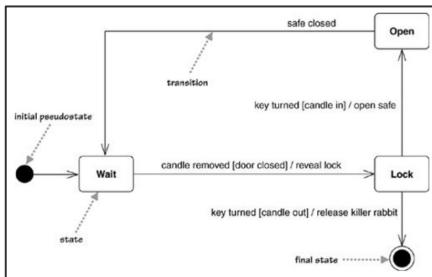
Deployment



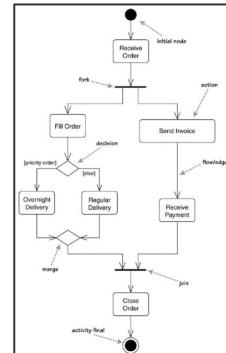
Use Case



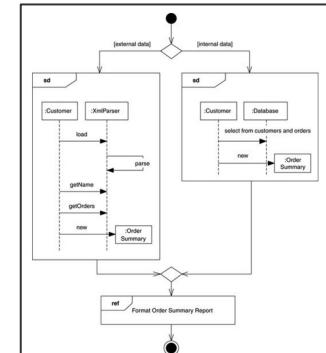
State



Activity

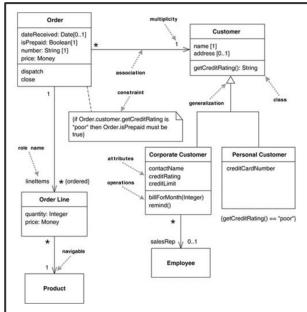


Interaction

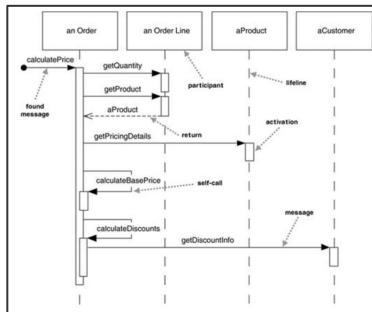


# UML: Unified Modeling Language

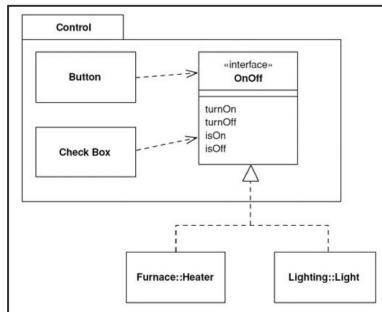
Class



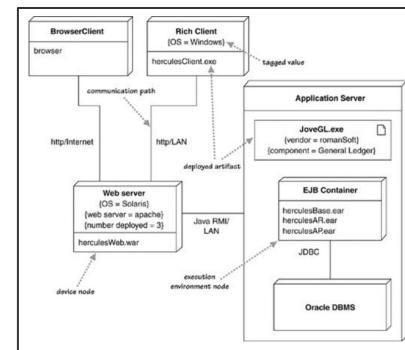
Sequence



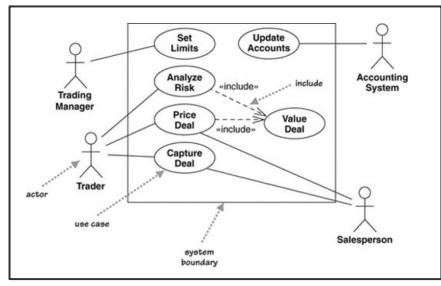
Package



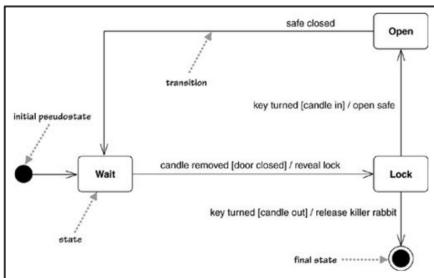
Deployment



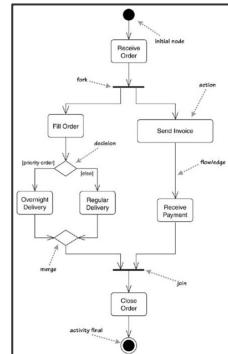
Use Case



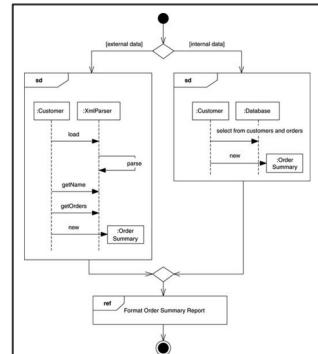
State



Activity

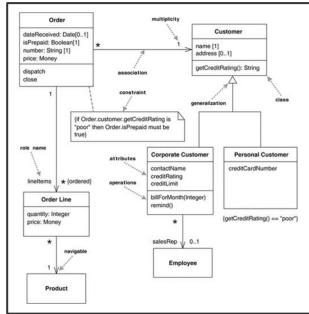


Interaction

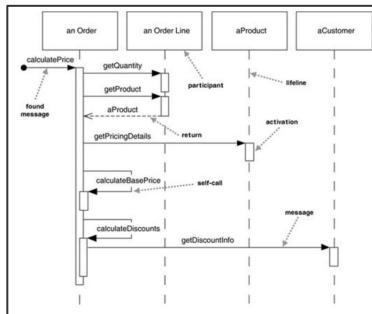


# UML: Unified Modeling Language

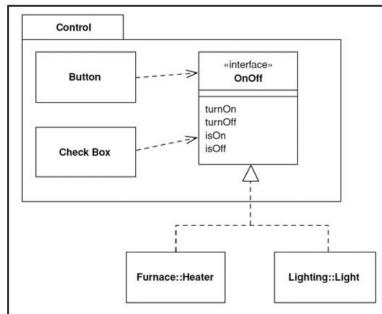
Class



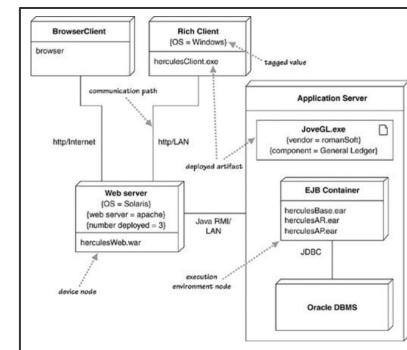
Sequence



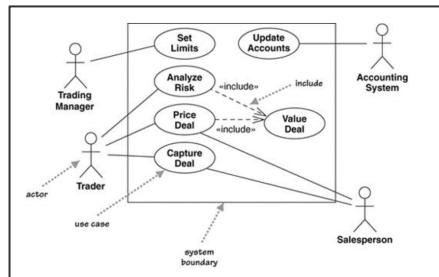
Package



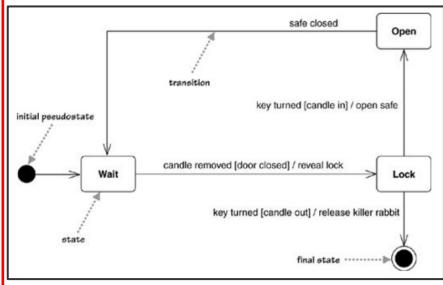
Deployment



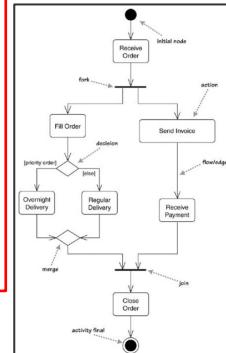
Use Case



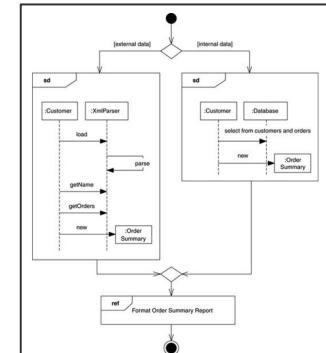
State



Activity

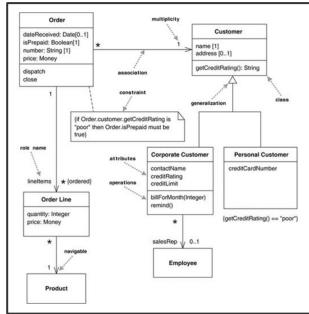


Interaction

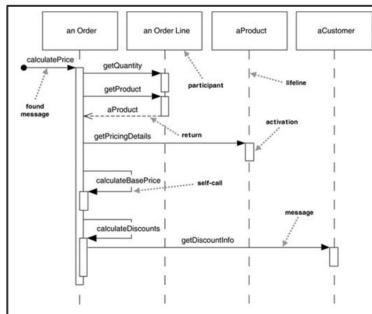


# UML: Unified Modeling Language

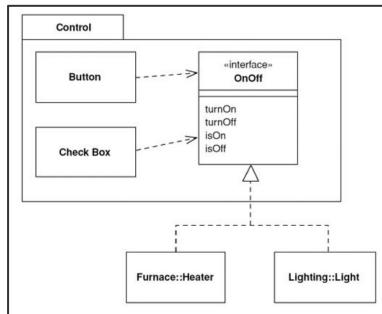
Class



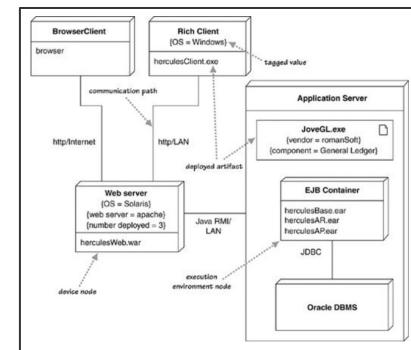
Sequence



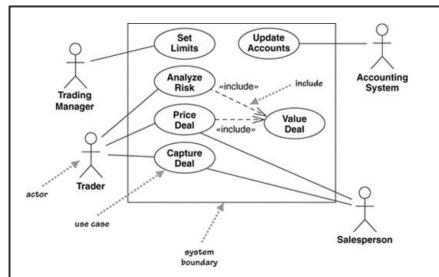
Package



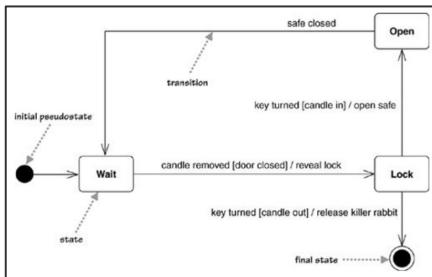
Deployment



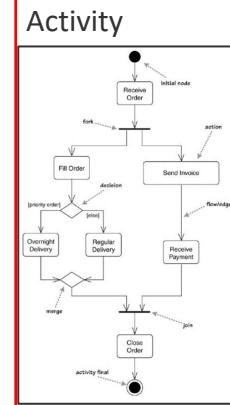
Use Case



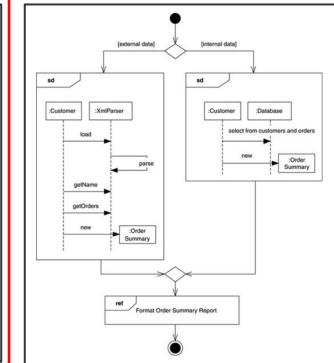
State



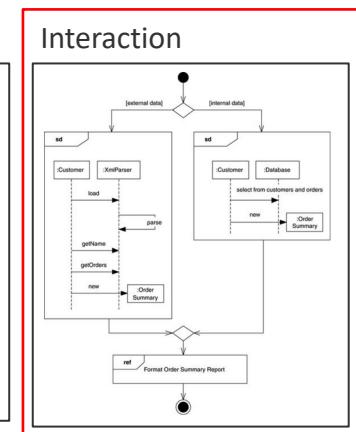
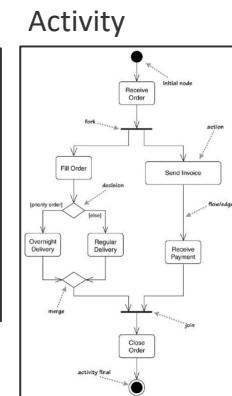
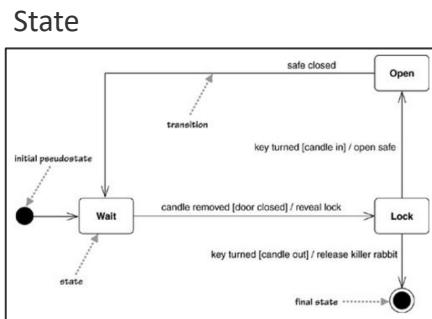
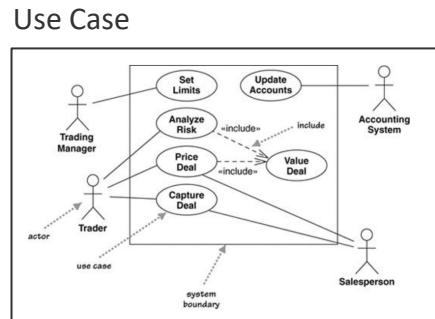
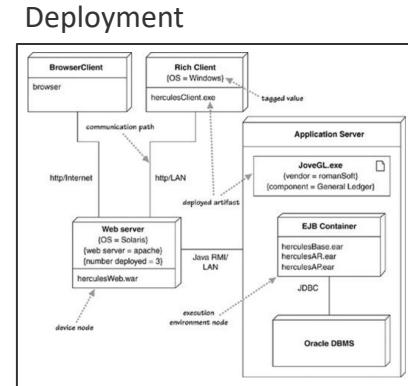
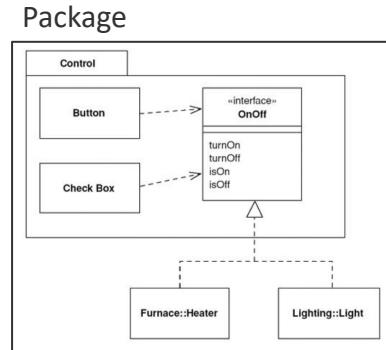
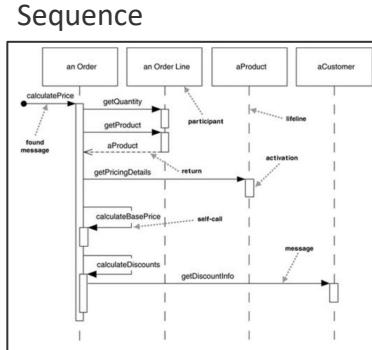
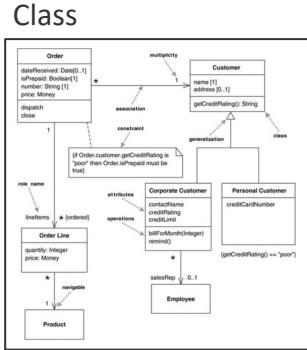
Activity



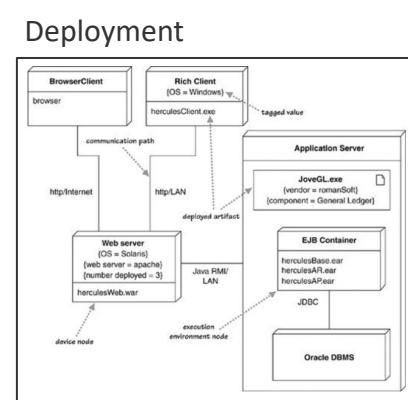
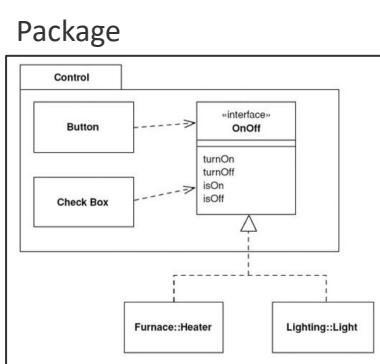
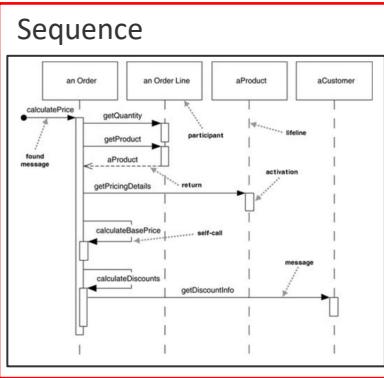
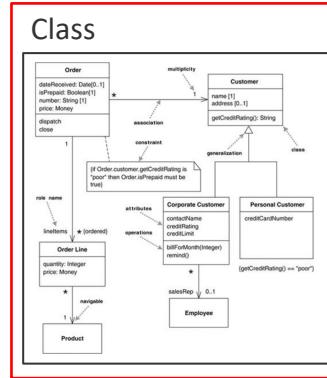
Interaction



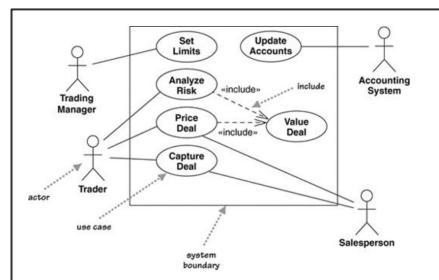
# UML: Unified Modeling Language



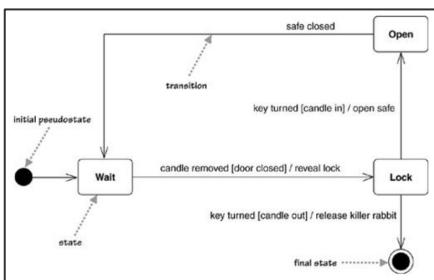
# UML: Unified Modeling Language



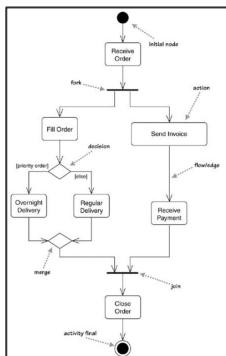
## Use Case



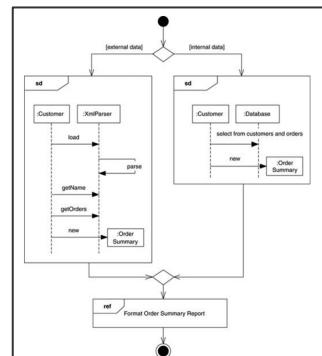
## State



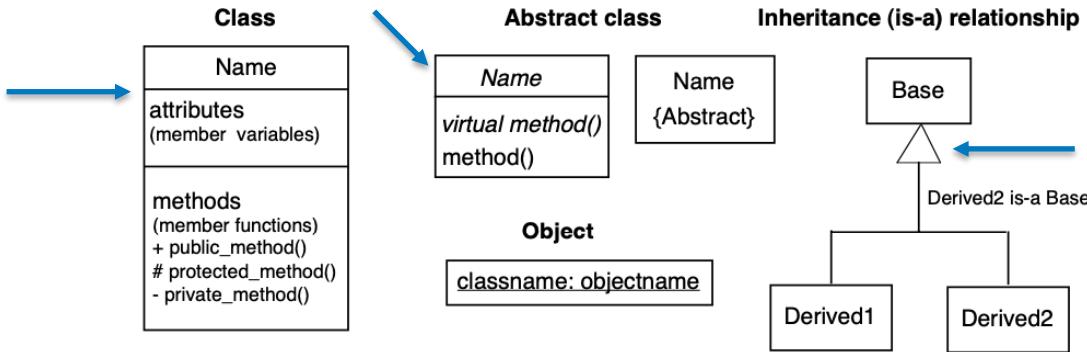
## Activity



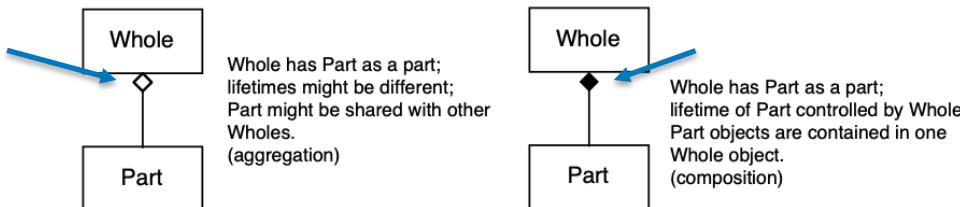
## Interaction



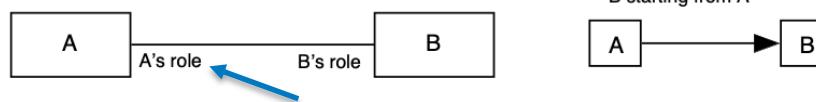
# UML: Class Diagram Model



## Aggregation and Composition (has-a) relationship

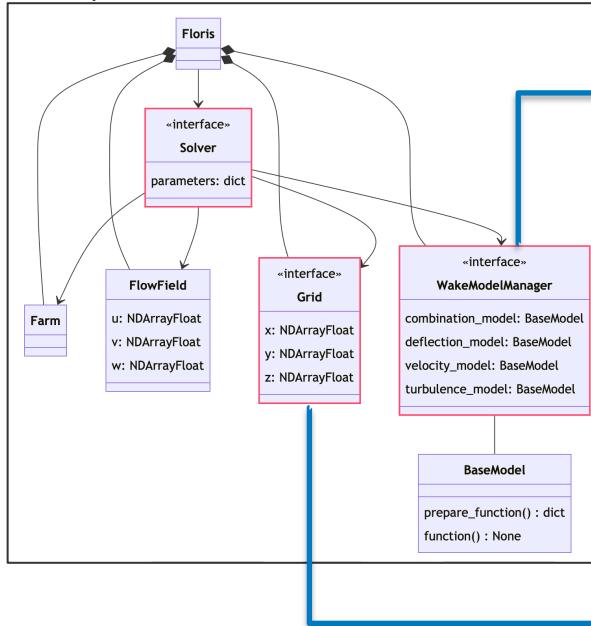


## Association (uses, interacts-with) relationship

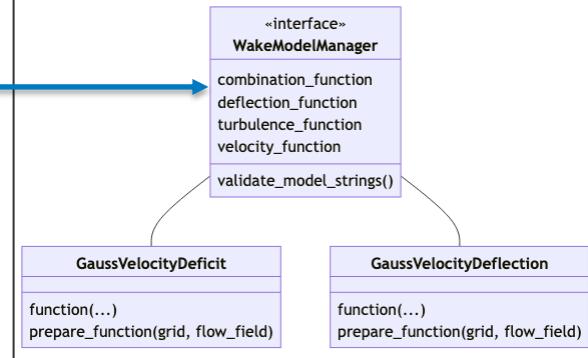


# UML: A note on perspective

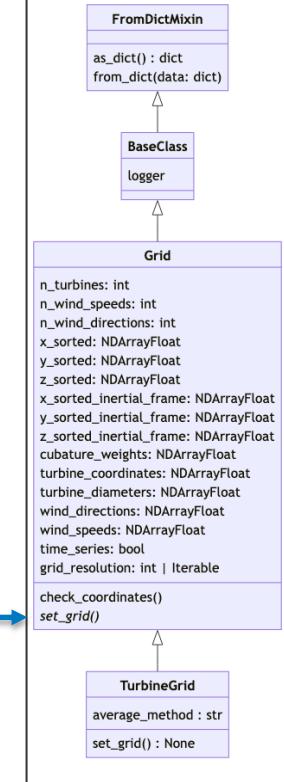
Conceptual



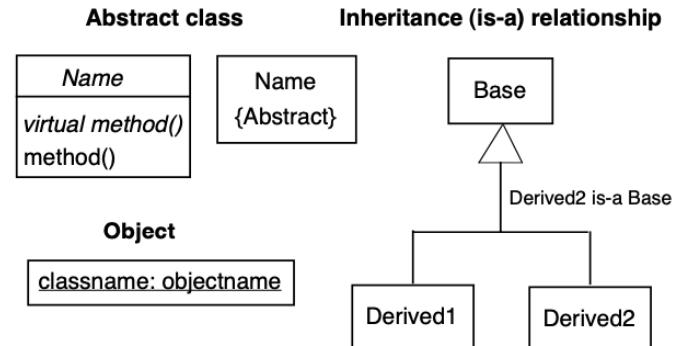
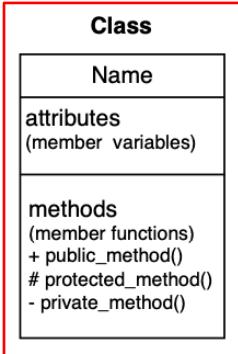
Specification



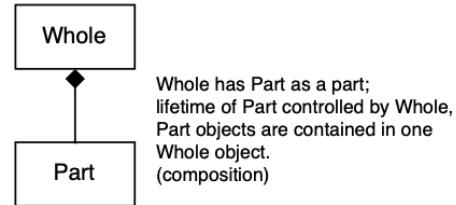
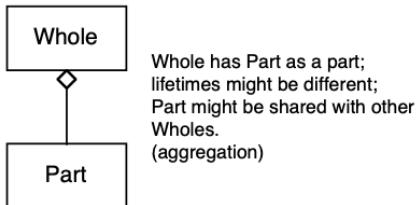
Implementation



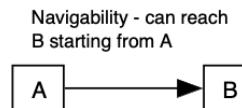
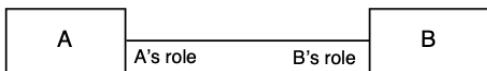
# UML: Class Diagram



### Aggregation and Composition (has-a) relationship



### Association (uses, interacts-with) relationship



## Example

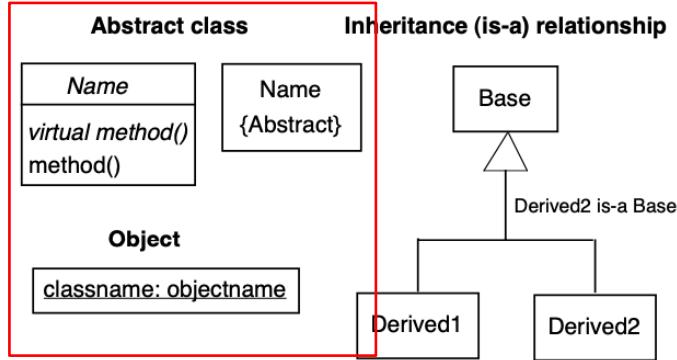
```

class WCompFloris:
    LEGEND : str
    LINE_PLOT_COLOR : str
    LINE_PLOT_LINESTYLE : str
    LINE_PLOT_MARKER : str
    deflection_model_string : str
    deflection_parameters : Optional[dict]
    fi : FlorisInterface
    floris_dict : dict
    hub_height
    rotor_diameter
    velocity_deficit_model_string : str
    velocity_deficit_parameters : dict
    yaw_angles : ndarray

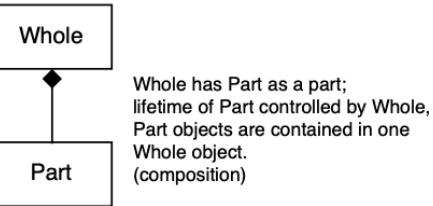
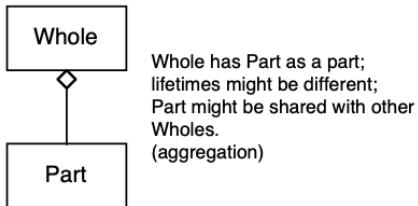
    AEP()
    horizontal_contour(wind_direction: float, resolution: tuple) : WakePlane
    streamwise_profile_plot(wind_direction: float, y_coordinate: float, xmin: float, xmax: float)
    vertical_profile_plot(wind_direction: float, x_coordinate: float, y_coordinate: float, zmax: float)
    xsection_contour(wind_direction: float, resolution: tuple, x_coordinate: float) : WakePlane
    xsection_profile_plot(wind_direction: float, x_coordinate: float, ymin: float, ymax: float)
  
```

# UML: Class Diagram

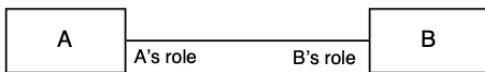
Class
Name
attributes (member variables)
methods (member functions) + public_method() # protected_method() - private_method()



## Aggregation and Composition (has-a) relationship



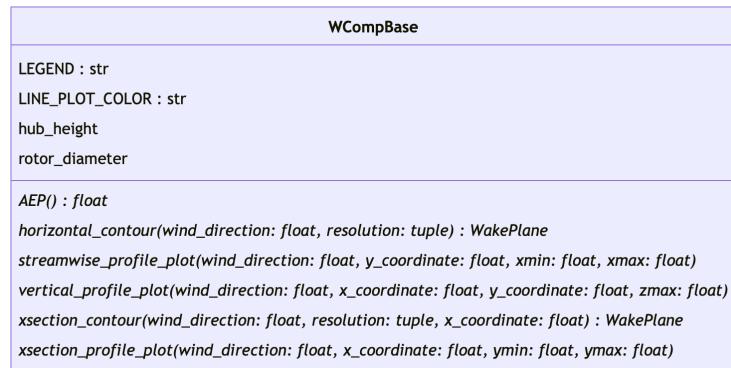
## Association (uses, interacts-with) relationship



Navigability - can reach B starting from A



## Example

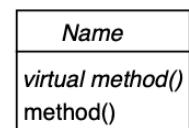


# UML: Class Diagram

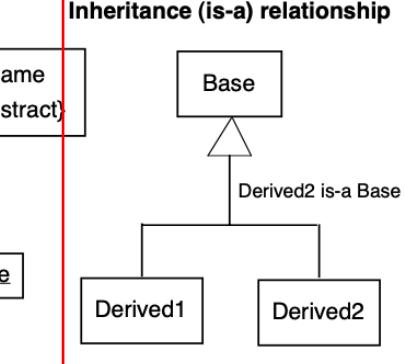
## Class

Name
attributes (member variables)
methods (member functions) + public_method() # protected_method() - private_method()

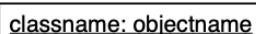
## Abstract class



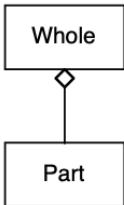
## Inheritance (is-a) relationship



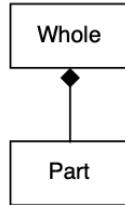
## Object



## Aggregation and Composition (has-a) relationship



Whole has Part as a part;  
lifetimes might be different;  
Part might be shared with other  
Wholes.  
(aggregation)



Whole has Part as a part;  
lifetime of Part controlled by Whole,  
Part objects are contained in one  
Whole object.  
(composition)

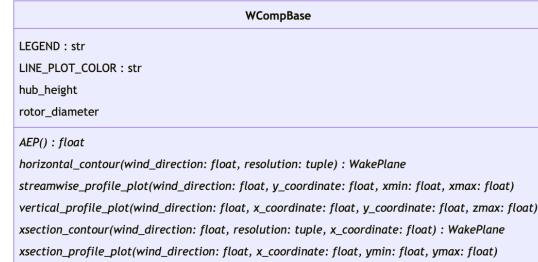
## Association (uses, interacts-with) relationship



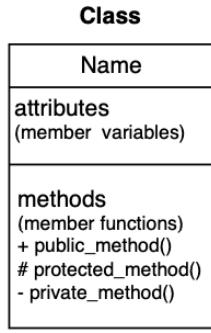
Navigability - can reach  
B starting from A



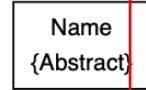
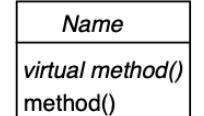
## Example



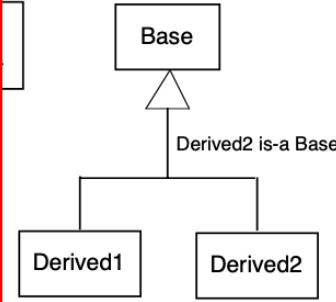
# UML: Class Diagram



**Abstract class**



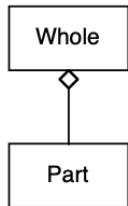
**Inheritance (is-a) relationship**



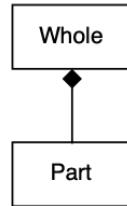
Object

classname: objectname

**Aggregation and Composition (has-a) relationship**

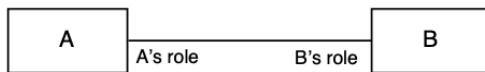


Whole has Part as a part;  
lifetimes might be different;  
Part might be shared with other  
Wholes.  
(aggregation)



Whole has Part as a part;  
lifetime of Part controlled by Whole,  
Part objects are contained in one  
Whole object.  
(composition)

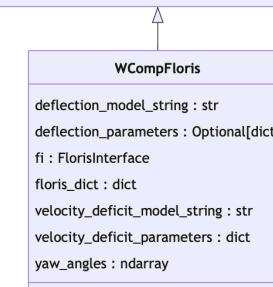
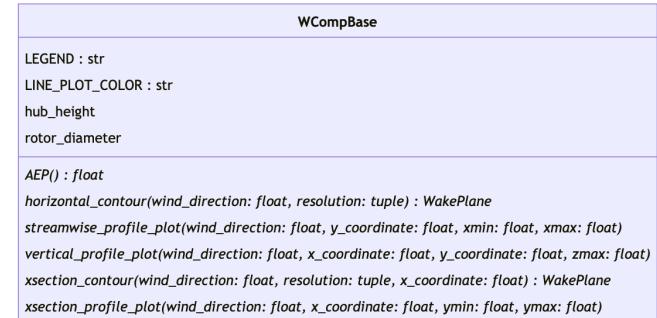
**Association (uses, interacts-with) relationship**



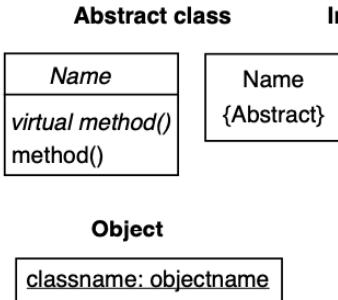
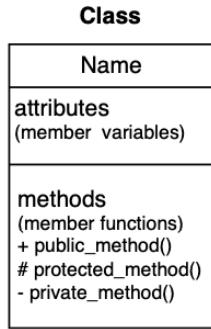
Navigability - can reach  
B starting from A



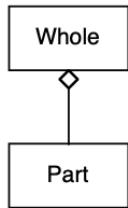
## Example



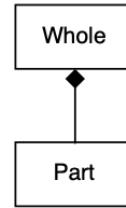
# UML: Class Diagram



## Aggregation and Composition (has-a) relationship

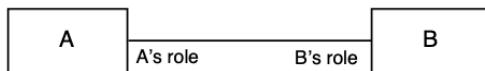


Whole has Part as a part;  
lifetimes might be different;  
Part might be shared with other  
Wholes.  
(aggregation)



Whole has Part as a part;  
lifetime of Part controlled by Whole,  
Part objects are contained in one  
Whole object.  
(composition)

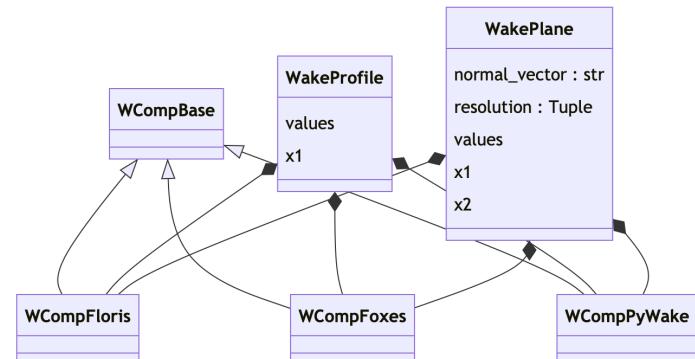
## Association (uses, interacts-with) relationship



Navigability - can reach  
B starting from A

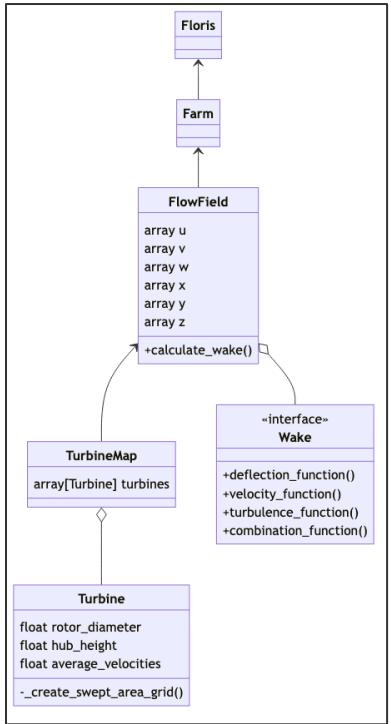


## Example

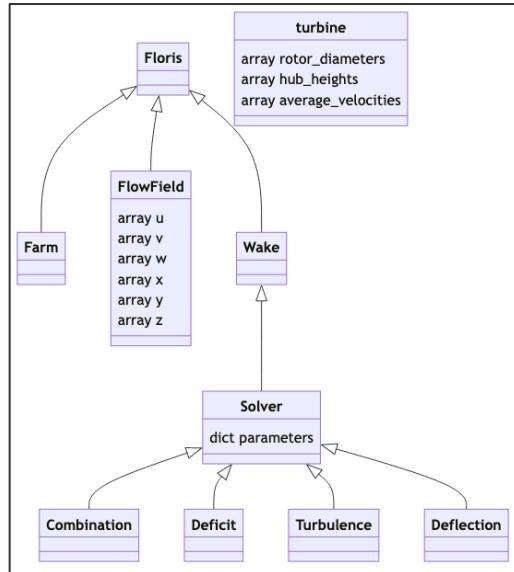


# UML: Class Diagram – In Design

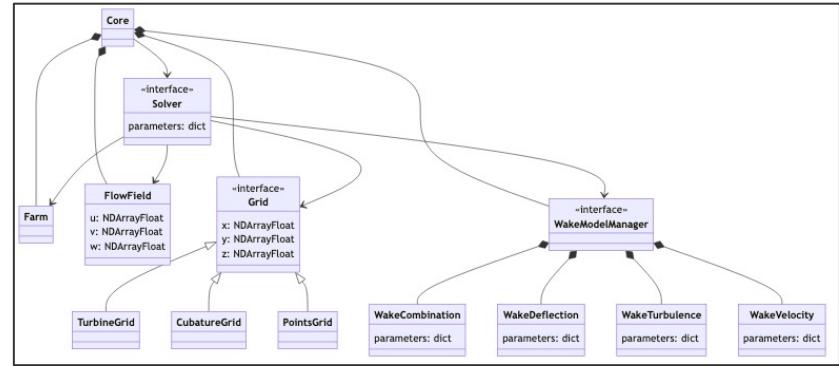
FLORIS v2



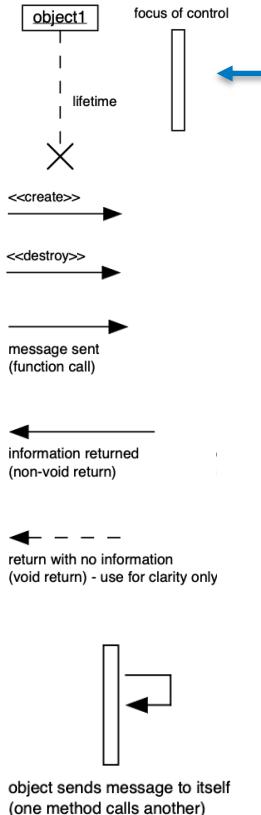
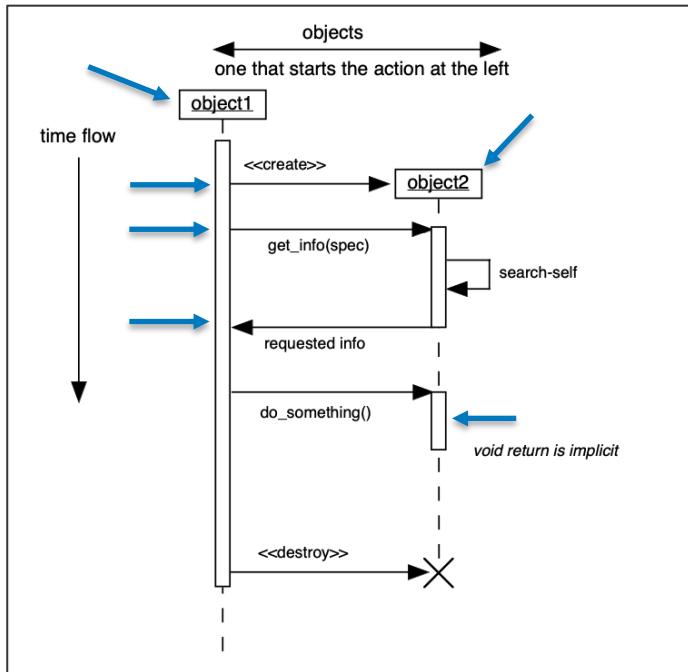
FLORIS v3



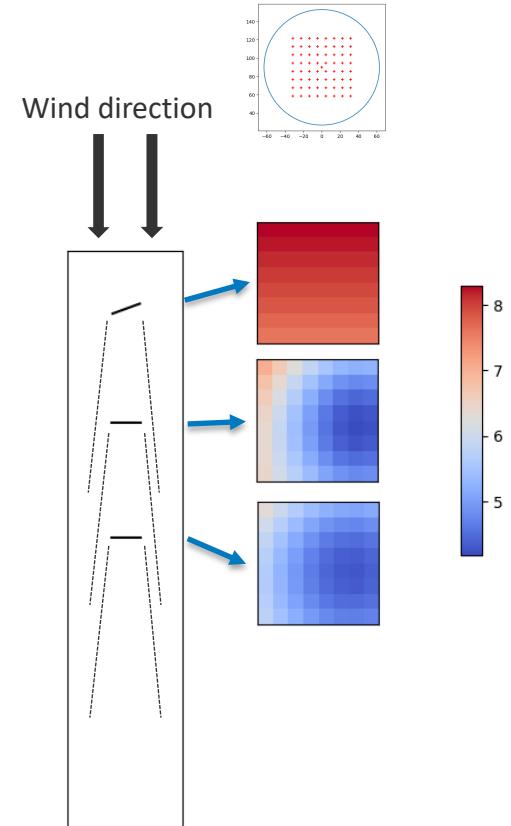
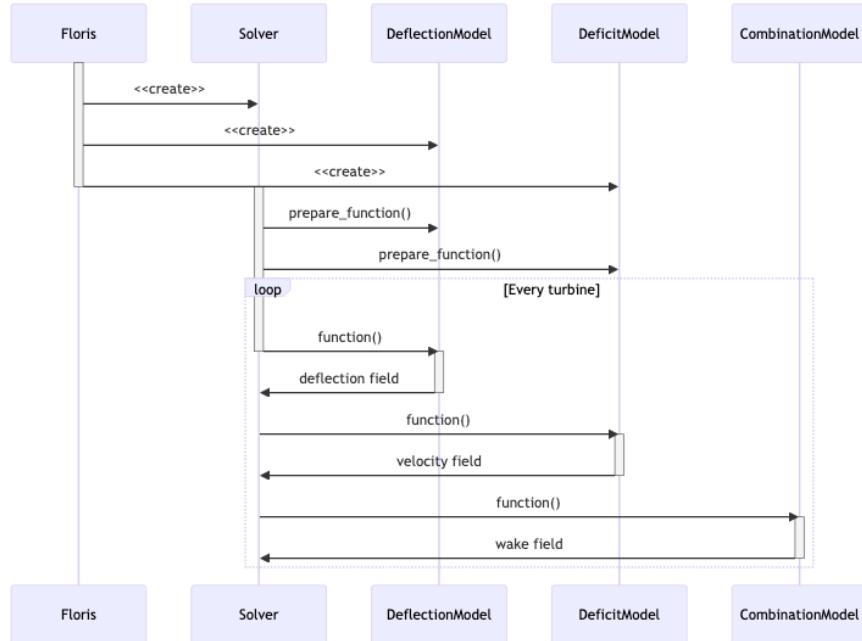
FLORIS v4



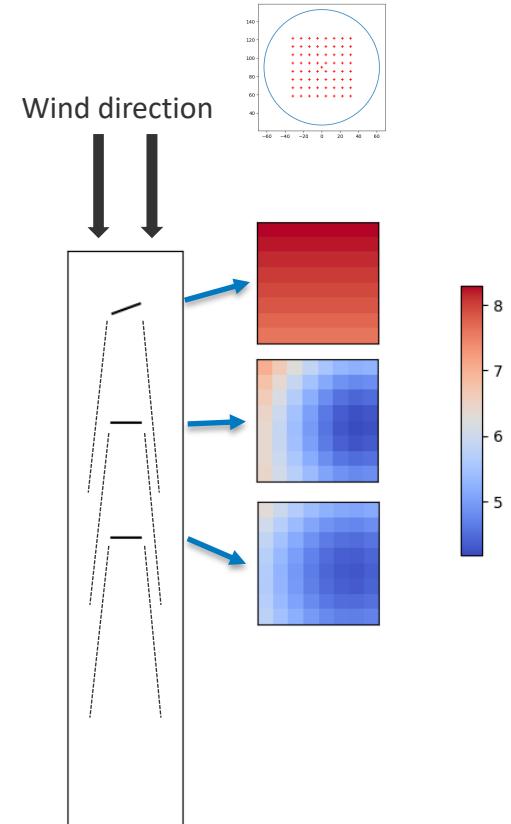
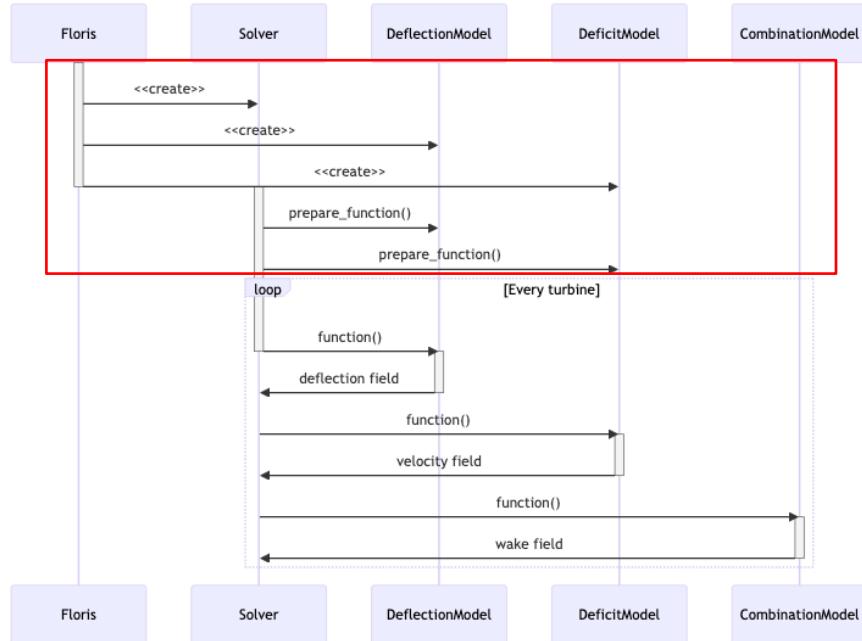
# UML: Sequence Diagram Model



# UML: Sequence Diagram

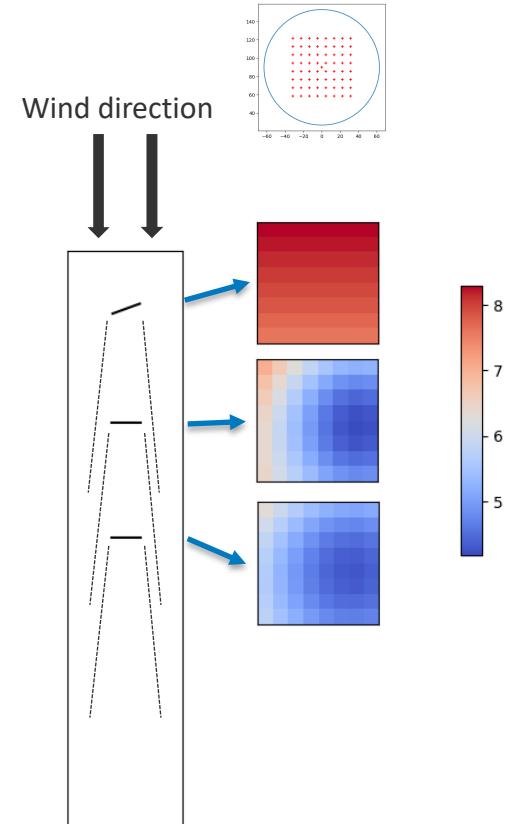
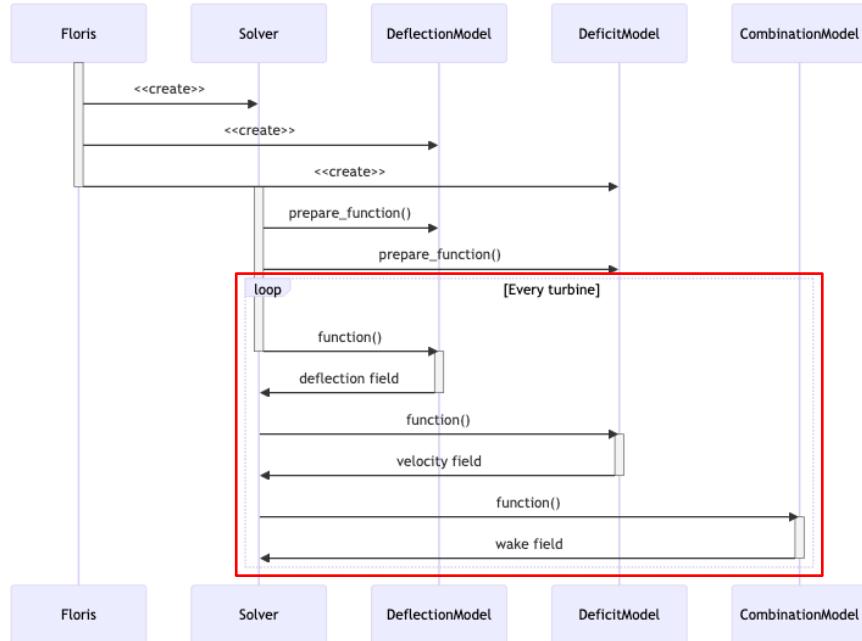


# UML: Sequence Diagram

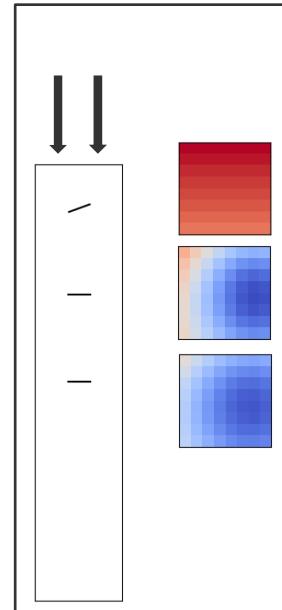
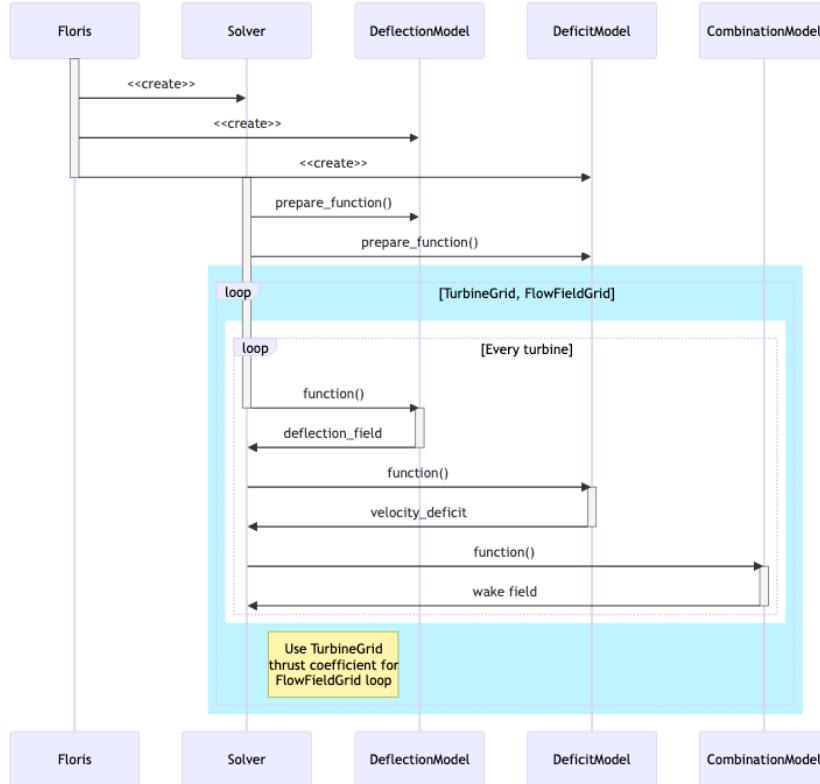


Rotor grid only

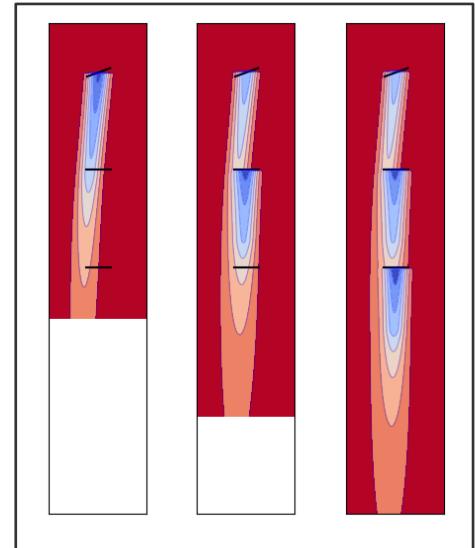
# UML: Sequence Diagram



# UML: Sequence Diagram

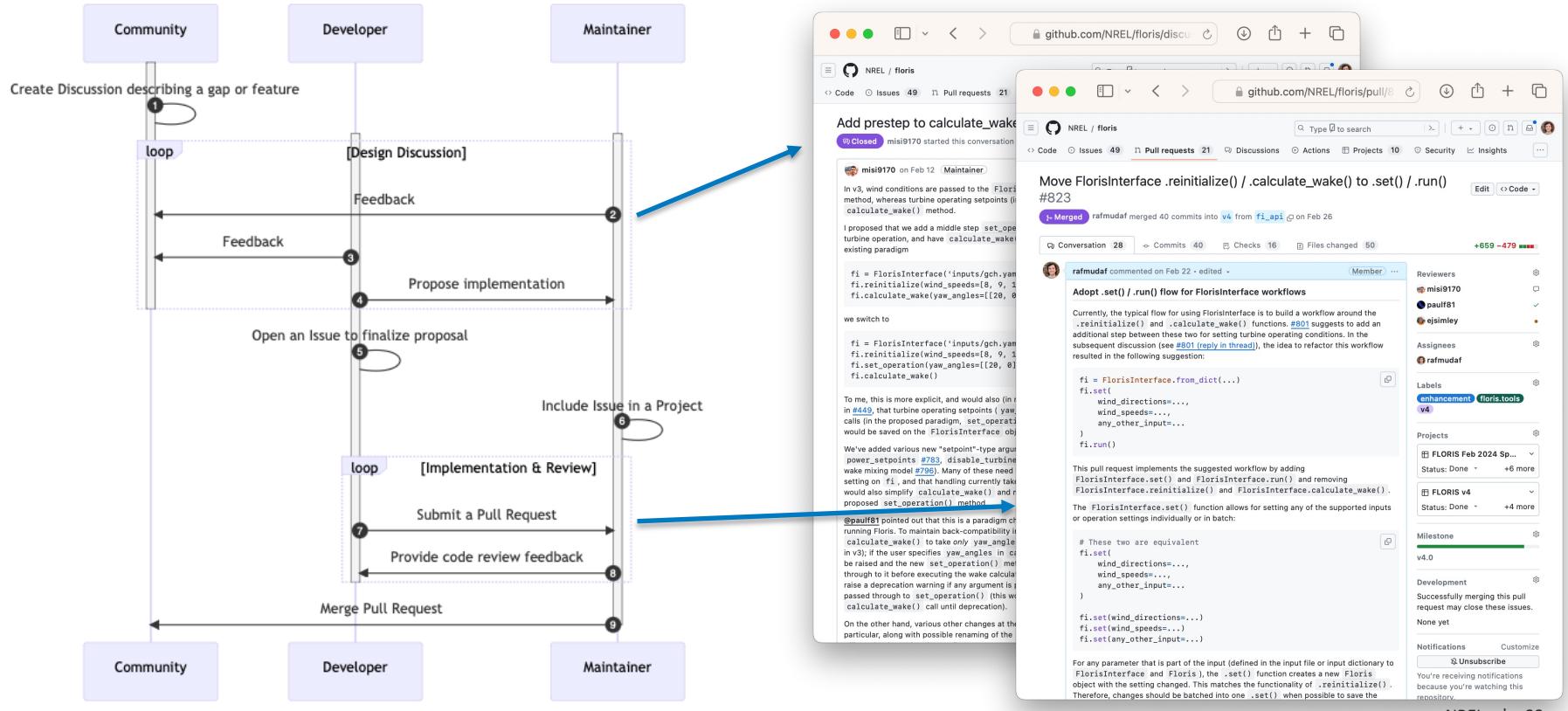


Inner loop

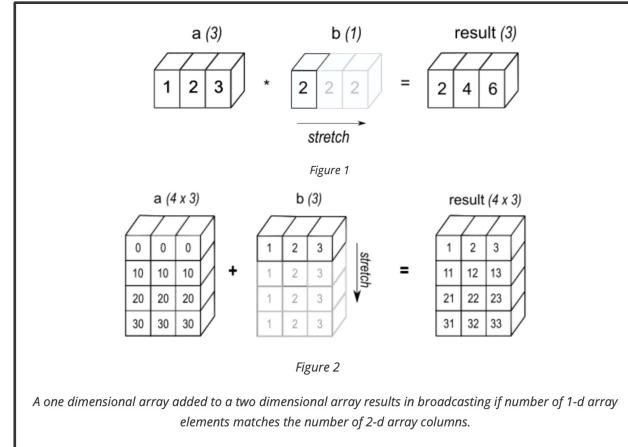
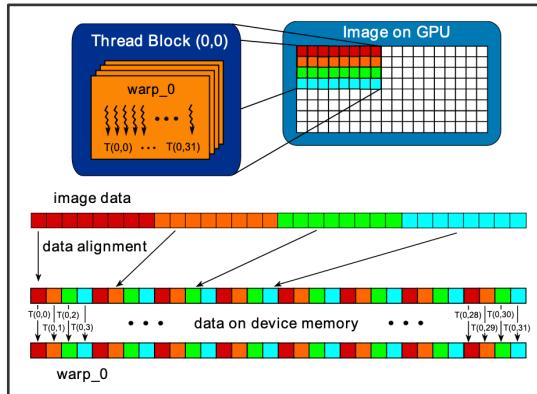
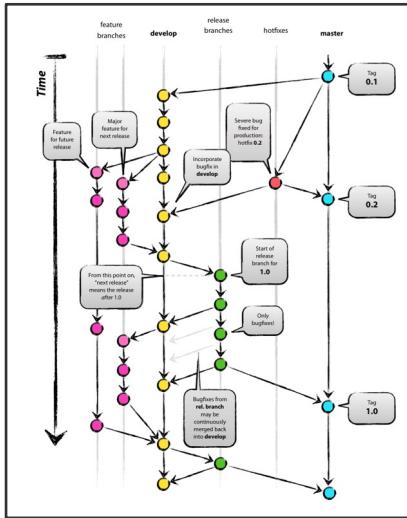


Outer loop

# UML: Sequence Diagram



# Other diagrams outside of UML



Use UML and anything else to get your message out there!

# Diagramming in the development workflow

---

# Documentation Driven Development

Documentation as Code (*Docs as Code*) refers to a philosophy that you should be writing documentation with the same tools as code:

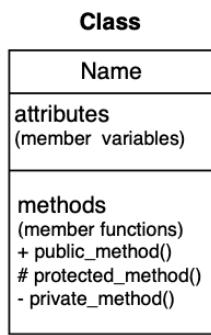
- Issue Trackers
- Version Control (Git)
- Plain Text Markup (Markdown, reStructuredText, Asciidoc)
- Code Reviews
- Automated Tests

This means following the same workflows as development teams, and being integrated in the product team. It enables a culture where writers and developers both feel ownership of documentation, and work together to make it as good as possible.

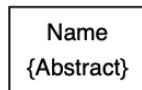
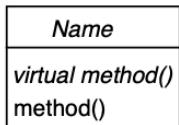
*Write the Docs*

<https://www.writethedocs.org/guide/docs-as-code/>

# UML: Class Diagrams



## Abstract class



## Inheritance (is-a) relationship



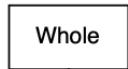
Derived2 is-a Base

Object

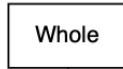
classname: objectname



## Aggregation and Composition (has-a) relationship



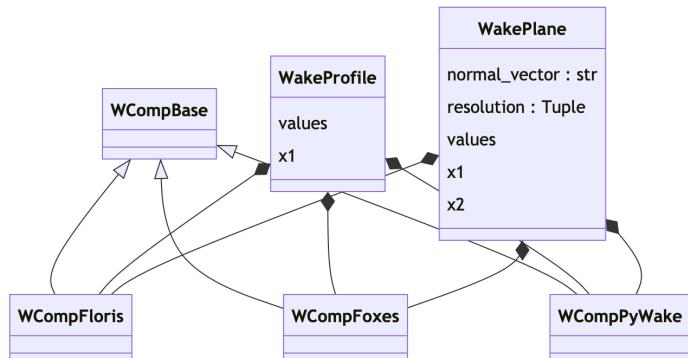
Whole has Part as a part;  
lifetimes might be different;  
Part might be shared with other  
Wholes.  
(aggregation)



Whole has Part as a part;  
lifetime of Part controlled by Whole,  
Part objects are contained in one  
Whole object.  
(composition)

## Example

INCORRECT



## Association (uses, interacts-with) relationship



Navigability - can reach  
B starting from A



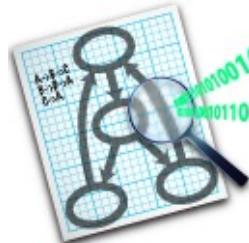
# In Practice: Automated Tools



Treat diagrams like other software infrastructure:

- Incorporate it into the development process
- Build tools, trust them, and lean on them heavily
- Automate – let the computers do the hard work

**doxygen**



Graphviz



Mermaid

linting    pylint

pyreverse

# In Practice: Doxygen

<https://www.doxygen.nl>

Code Documentation. Automated.

Free, open source, cross-platform.

Doxygen

Main Page Related Pages Namespaces Classes Files Search

+ find()

template<typename K, typename V>  
V\* Cache<K, V>::find ( const K & key )

Finds a value in the cache given the corresponding key.

Returns

a pointer to the value or nullptr if the key is not found in the cache

Note

The hit and miss counters are updated, see `hits()` and `misses()`.

Definition at line 105 of file cache.h.

```
106     {
107         auto it = m_cachetItemMap.find(key);
108         if (it != m_cachetItemMap.end())
109         {
110             // move the item to the front of the list
111             m_cachetItemList.splice(m_cachetItemList.begin(), it->second);
112             m_hits++;
113             // return the value
114             return it->second;
115         }
116         else
117         {
118             m_misses++;
119         }
120     }
121     return nullptr;
122 }
```

References `Cache<K, V>::m_cachetItemList`, `Cache<K, V>::m_cachetItemMap`, `Cache<K, V>::m_hits`, and `Cache<K, V>::m_misses`.

Generated by doxygen 1.10.0

Show dark mode output

Doxygen is a widely-used documentation generator tool in software development. It automates the generation of documentation from source code comments, parsing information about classes, functions, and variables to produce output in formats like HTML and PDF. By simplifying and standardizing the documentation process, Doxygen enhances collaboration and maintenance across diverse programming languages and project scales.



Multiple formats



Markdown



C++



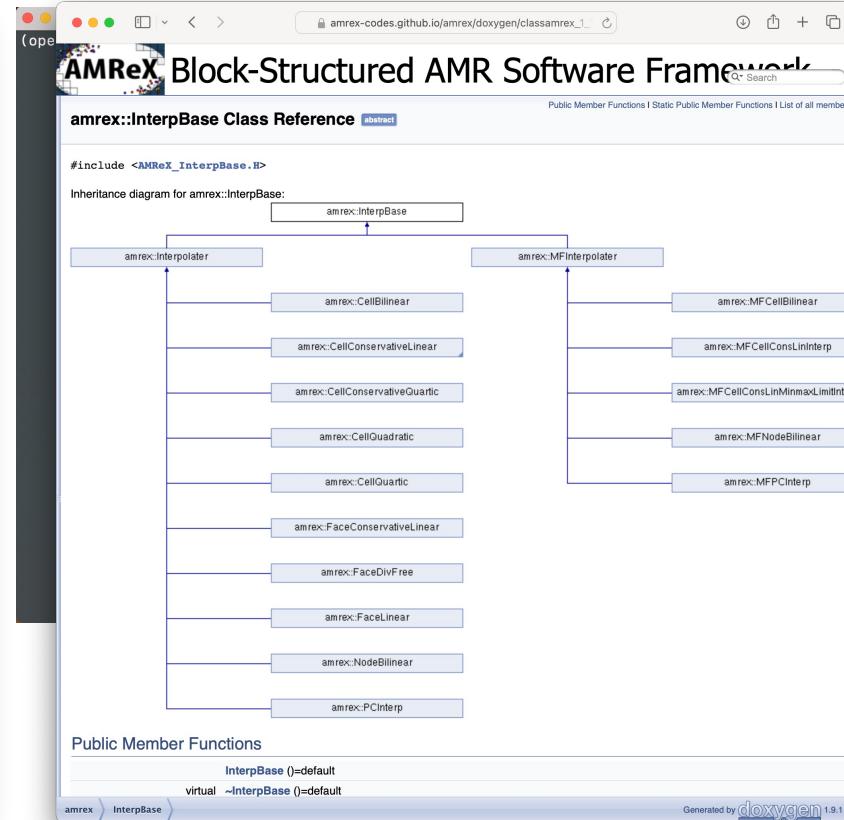
Cross-referencing



Diagrams



Configuration



# In Practice: Mermaid

<https://mermaid.js.org>

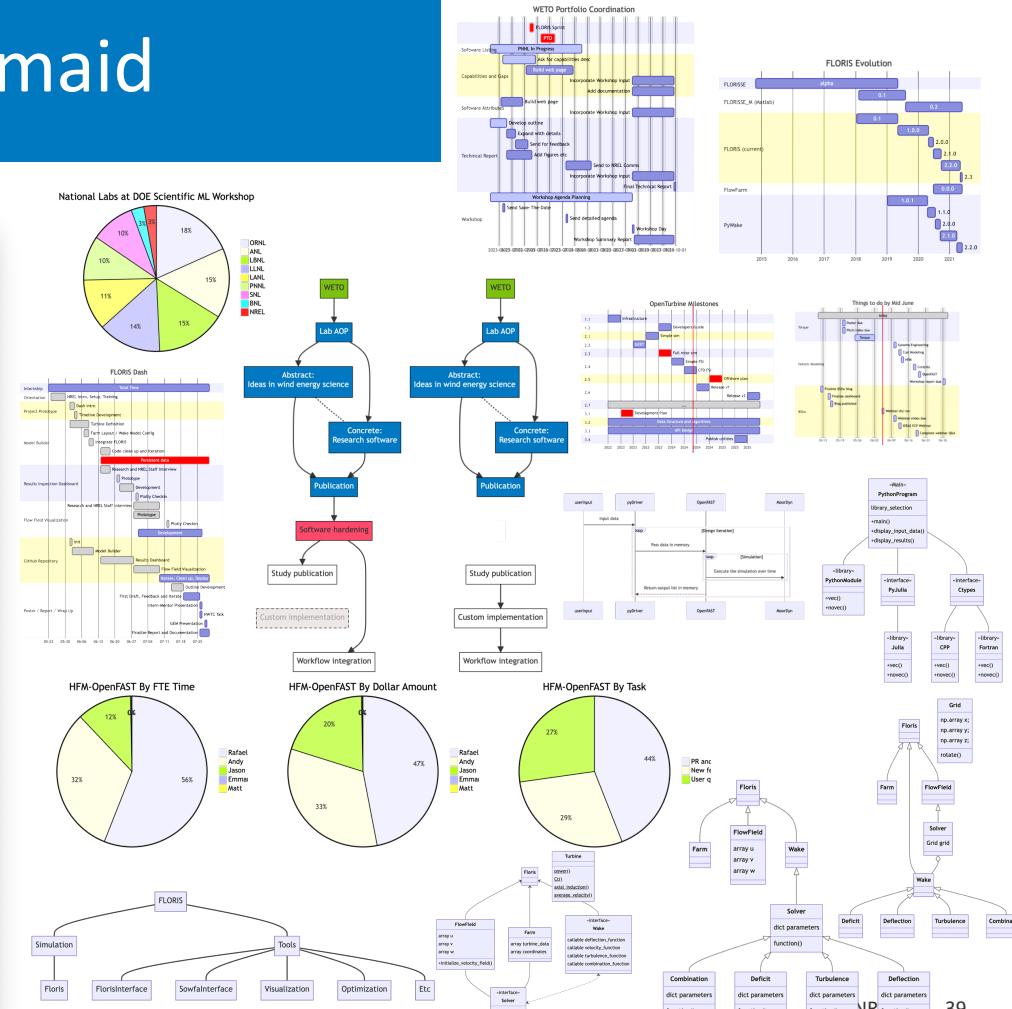
The Mermaid.js homepage features a large central logo with a white 'Y' shape on a pink-to-white gradient background. Below the logo, the text reads: "Mermaid Diagramming and charting tool". To the left, a box states: "JavaScript based diagramming and charting tool that renders Markdown-inspired text definitions to create and modify diagrams dynamically." Below this are two buttons: "Get Started" and "View on GitHub". At the bottom, there are four boxes: "Easy to use!" (with a plus icon), "Integrations available!" (with a clover icon), "Award winning!" (with a trophy icon), and "Mermaid + Mermaid Chart" (with a bar chart icon).

**Easy to use!**  
Easily create and render detailed diagrams and charts with the Mermaid Live Editor.

**Integrations available!**  
Use Mermaid with your favorite applications, check out the integrations list.

**Award winning!**  
2019 JavaScript Open Source Award winner for "The Most Exciting Use of Technology".

**Mermaid + Mermaid Chart**  
Mermaid Chart is a major supporter of the Mermaid project.



# In Practice: Mermaid

GitHub Discussion – same for any GitHub-flavored Markdown product

A screenshot of a GitHub discussion page. The URL is [github.com/IEAWindTask37/windIO/discussions/23](https://github.com/IEAWindTask37/windIO/discussions/23). The title of the discussion is "Scope, architecture, and dependency structure #23". The post was made by rafmudaf on Oct 24, 2023. The content area contains a Mermaid class diagram:

```
```mermaid
classDiagram
    windIO *-- loader
    windIO *-- Plant
    windIO *-- Turbine
    class loader{
        +dict load()
        +validate()
    }
    class Plant{
        +common
        +energy_resource
        +site
        +turbine
        +wind_energy_system
        +wind_farm
    }
    class Turbine{
        +ontology
    }
```


The Mermaid code defines a windIO object with three associations: loader, Plant, and Turbine. Each association is marked with a multiplicity of * and a dashed line. Below the code, there is a preview of the diagram and various GitHub-style UI elements like a toolbar, sidebar, and footer buttons.


```

A screenshot of the same GitHub discussion page after the Mermaid code has been processed. The title is now "Scope, architecture, and dependency structure #23". The post by rafmudaf is still visible. The content area now displays the generated class diagram:

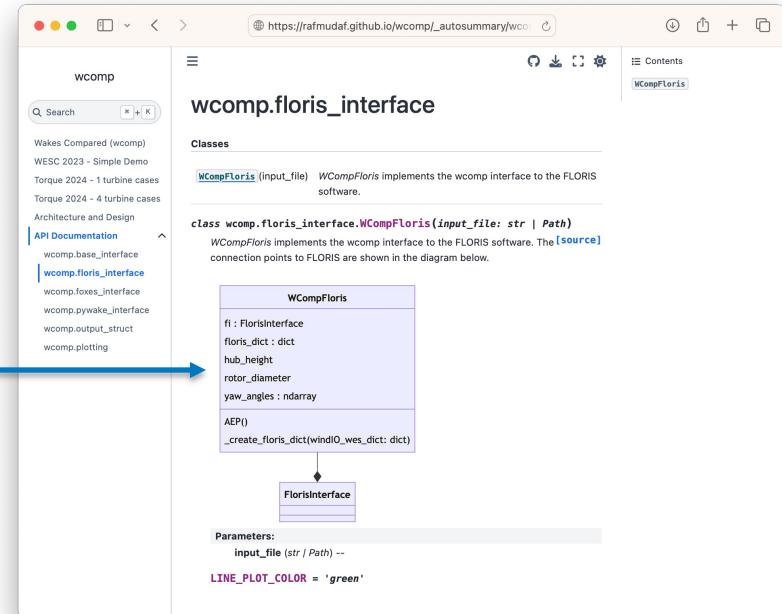
```
graph TD
    windIO[windIO] *--> loader[loader]
    windIO[windIO] *--> Plant[Plant]
    windIO[windIO] *--> Turbine[Turbine]
    class loader{
        +dict load()
        +validate()
    }
    class Plant{
        +common
        +energy_resource
        +site
        +turbine
        +wind_energy_system
        +wind_farm
    }
    class Turbine{
        +ontology
    }

```

The diagram shows a `windIO` object at the top, connected to three other objects: `loader`, `Plant`, and `Turbine`. Each of these three objects has its own associated class definition below it. The `Plant` class definition includes methods `+common`, `+energy_resource`, `+site`, `+turbine`, `+wind_energy_system`, and `+wind_farm`. The `Turbine` class definition includes the method `+ontology`. The GitHub interface includes standard navigation and search tools.

## In Practice: Mermaid

Sphinx-based documentation – add diagrams anywhere including in API docs



# In Practice: pyreverse

<https://pylint.readthedocs.io/en/latest/pyreverse.html>

Pylint 3.3.0-dev0 documentation

Pyreverse

Pyreverse analyzes your source code and generates package and class diagrams.

It supports output to .dot/.gv, .puml/.plantuml (PlantUML) and .mmd/.html (MermaidJS) file formats. If Graphviz (or the dot command) is installed, all output formats supported by Graphviz can be used as well. In this case, pyreverse first generates a temporary .gv file, which is then fed to Graphviz to generate the final image.

## Running Pyreverse

To run pyreverse, use:

```
pyreverse [options] <packages>
```

<packages> can also be a single Python module. To see a full list of the available options, run:

```
pyreverse -h
```

## Example Output

Example diagrams generated with the .puml output format are shown below.

### Class Diagram

```
graph TD; DoNothing --> Specialization; DoNothing2 --> Specialization; Ancestor --> Specialization; Interface --> Ancestor
```

```
(floris) >>> $ pyreverse --help
usage: pyreverse [options]

Create UML diagrams for classes and modules in <packages>.

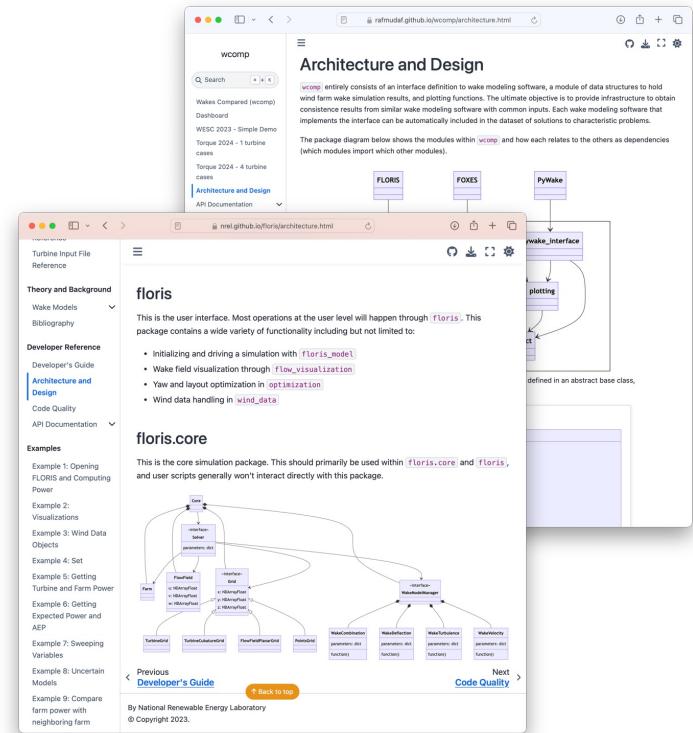
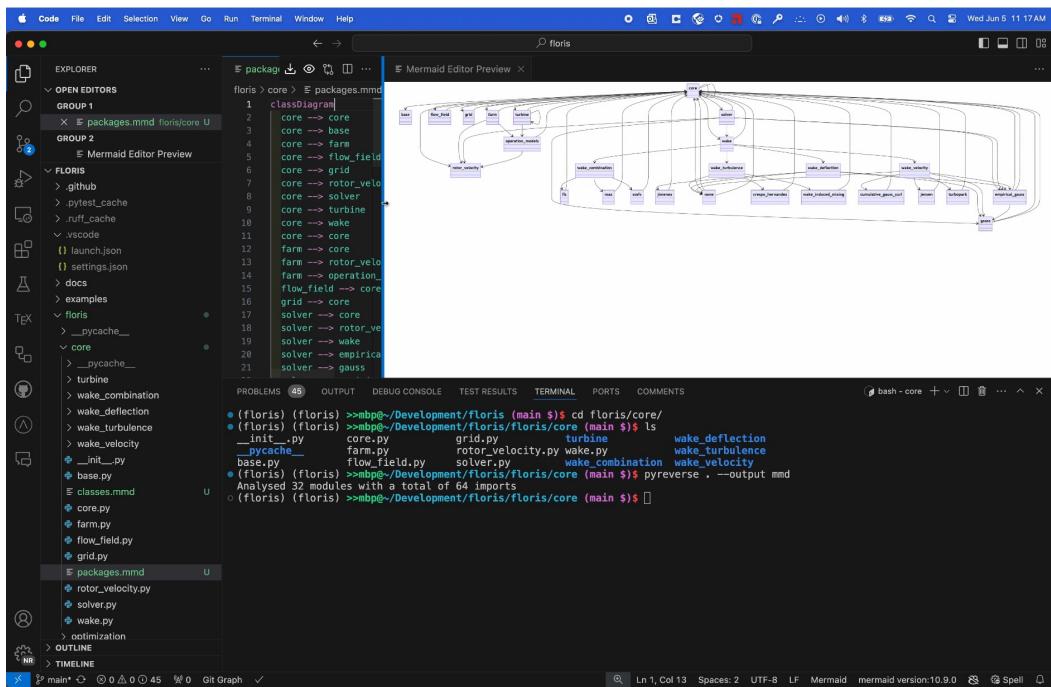
options:
  -h, --help            show this help message and exit

Pyreverse:
  Base class providing common behaviour for pyreverse commands.

--filter-mode <mode>, -f <mode>
  filter attributes and functions according to <mode>. Correct modes are: 'PUB_ONLY' filter all non public attributes
  [DEFAULT], equivalent to PRIVATE+SPECIAL_A 'ALL' no filter 'SPECIAL' filter Python special functions except constructor
  'OTHER' filter protected and private attributes (default: PUB_ONLY)
--class <class>, -c <class>
  create a class diagram with all classes related to <class>; this uses by default the options -ASmy (default: [])
--show-ancestors <ancestor>, -a <ancestor>
  show <ancestor> generations of ancestor classes not in <projects> (default: None)
--all-ancestors, -A
  show all ancestors off all classes in <projects> (default: None)
--show-associated <association_level>, -s <association_level>
  show <association_level> levels of associated classes not in <projects> (default: None)
--all-associated, -S
  show recursively all associated off all associated classes (default: None)
--show-builtins, -b
  include builtin objects in representation of classes (default: False)
--show-stdlib, -L
  include standard library objects in representation of classes (default: False)
--module-names <y or n>, -m <y or n>
  include module name in representation of classes (default: None)
--only-classnames, -k
  don't show attributes and methods in the class boxes; this disables -f values (default: False)
--no-standalone
  only show nodes with connections (default: False)
--output <format>, -o <format>
  create a *.<format> output file if format is available. Available formats are: dot, puml, plantuml, mmd, html. Any
  other format will be tried to create by means of the 'dot' command line tool, which requires a graphviz installation.
  (default: dot)
--colorized
  Use colored output. Classes/modules of the same package get the same color. (default: False)
--max-color-depth <depth>
  Use separate colors up to package depth of <depth> (default: 2)
--color-palette <color1,color2,...>
  Comma separated list of colors to use (default: ('#77AADD', '#99DDFF', '#44BB99', '#BBBB33', '#AAAA00', '#EEDD88',
  '#EE8866', '#FFAA8B', '#DDDDDD'))
--ignore <file[,file...]>
  Files or directories to be skipped. They should be base names, not paths. (default: ('CVS',))
--project <project name>, -p <project name>
  set the project name. (default: )
--output-directory <output_directory>, -d <output_directory>
  set the output directory path. (default: )
--source-roots <path>[,spath...]
  Add paths to the list of the source roots. Supports globbing patterns. The source root is an absolute path or a path
  relative to the current working directory used to determine a package namespace for modules located under the source
  root. (default: ())
--verbose
  Makes pyreverse more verbose/talkative. Mostly useful for debugging. (default: False)
```

## In Practice: pyreverse

## Static analysis of Python source code to generate package and class diagrams



# In Practice: AppMap

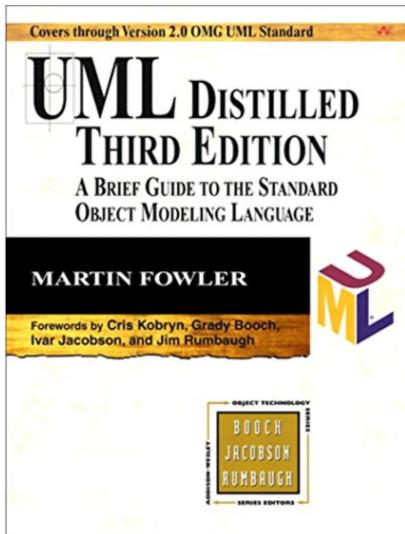
<https://appmap.io/product/appmap-in-the-code-editor>

The screenshot shows the AppMap product page with a dark theme. On the left, there's a large call-to-action button with the text "VISUALIZE YOUR RUNNING CODE". Below it, a subtext says "An open source personal observability platform." and a "Get AppMap" button. On the right, there's a screenshot of the AppMap interface showing a dependency map with various nodes like "logger", "LogDevice", "write", "array", "Loop", and "Integer". Below the map, there's a sequence diagram and some parameters and return value details.

The screenshot shows the AppMap plugin integrated into a code editor. On the left, the code editor interface is visible with tabs for "Code", "File", "Edit", etc. In the center, there's a "Dependency Map" view showing a network of objects and their interactions. On the right, there's a terminal window showing a test session starting with Python 3.10.13, pytest-8.2.2, and pluggy-1.5.0. The terminal output shows a test session starting with 1 passed, 3 deselected in 0.15s. The bottom part of the screen shows the AppMap interface with a detailed dependency graph and a sidebar with navigation links like "Ask Navie", "APPMAP DATA", "PACKAGES", "CLASSES", and "FUNCTIONS".

# Summary

1. Read this book



2. Use these tools

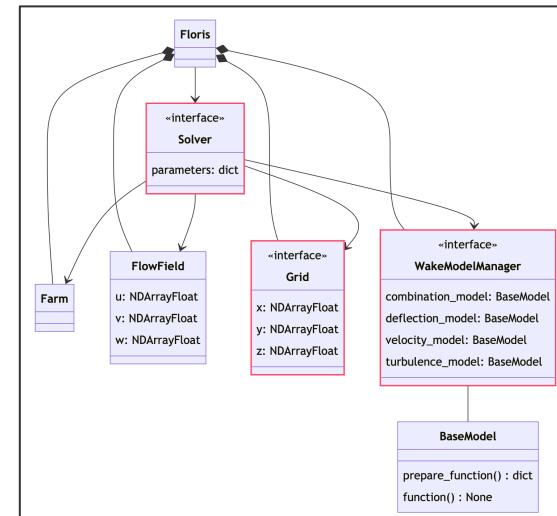


linting    pylint

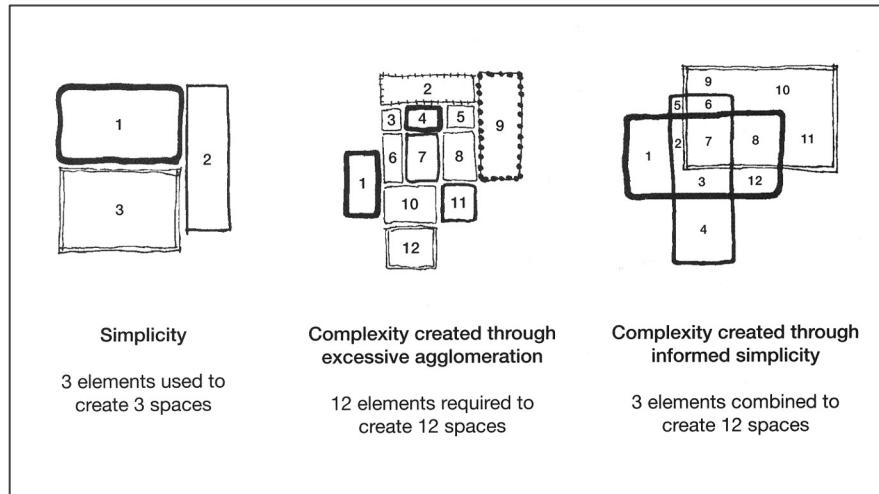


doxygen

3. Directly communicate your software design



# Stepping out to the big picture



*101 Things I Learned in Architecture School*



# Thank you!

[rafael.mudafort@nrel.gov](mailto:rafael.mudafort@nrel.gov)

@rafmudaf

[rafmudaf.github.io/communicating-design](https://rafmudaf.github.io/communicating-design)