Understand Graphs

Table of Contents

1. Assigned To
2. Assigned To & Assignments
3. Assignments
4. Base Classes
5. Butterfly
6. Called By
7. Calls
8. Calls & Called By
9. Child Lib Units
10. Child Lib Units & Parent Lib Unit
11. Contains
12. Control Flow
13. Data Components
14. Data Flow In
15. Data Flow Out
16. Data Flow Out & Data Flow In
17. Data Members
18. Declaration
19. Declaration File
20. Declaration Tree
21. Declaration Tree & Declared In
22. Declaration Type
23. Declared In
24. Depended On By
25. Dependencies
26. Depends On
27. Derived Classes
28. Derived Classes & Base Classes
29. Extended By
30. Extended By & Extends
31. Extends
32. File Dependencies
33. Graph Architecture
34. Imported By
35. Imports
36. Imports & Imported By
37. Include
38. Include & Includeby
39. Includeby
40. Instantiation

41. Instantiations
42. Object References
43. Overriddenby
44. Overriddenby & Overrides
45. Overrides
46. Parent Declaration
47. Parent Lib Unit
48. Rename Declaration
49. Required By
50. Requires
51. Requires & Required By
52. Return Type
53. Type Derived From
54. Type Tree
55. Type Tree & Type Derived From
56. UML Class Diagram
57. UML Sequence Diagram
58. Used By
59. Uses
60. Uses & Used By
61. Uses Traits
62. Uses Traits & Used By
63. Variable Tracker
64. With
65. With & Withby
66. Withby

Assigned To

Languages: Ada, C#, C++, Java
Targets: Functions, Objects
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the objects, functions, and parameters that a selected entity's value is directly or indirectly assigned into.

For a selected object, variable, or parameter, this shows edges to the entities on the left-hand side of assignments where the selected entity appears on the right, then recurses forward into whatever those entities are themselves assigned into, building a multi-level tree. Passing the entity as a function argument counts as an assignment to the corresponding parameter, so edges continue through the parameter into whatever the function does with it; if the function's return value depends on the entity, the graph continues from the function into wherever that return value is used.

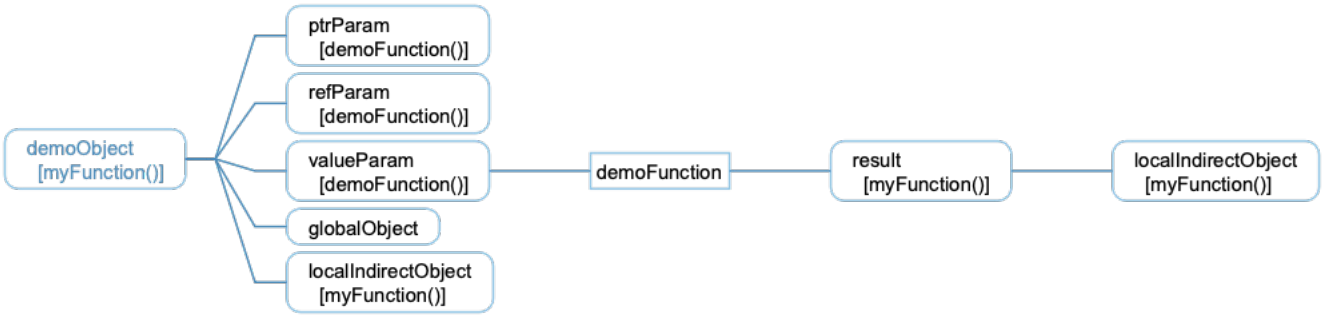
Edges are solid when the entity is assigned directly (e.g. globalObject = demoObject;) and dashed when it's only part of a larger expression (e.g. localIndirectObject = demoObject + result;). Compound assignment operators like += and ++ are tracked the same way — always indirectly, since they combine the existing value with another operand. The inverse graph, Assignments, shows the reverse relationship — the assignments a selected entity's value is built from.

For example, see the following code and corresponding graph

```
1 extern int globalObject;
2
3 int demoFunction(int valueParam, int &refParam, int *ptrParam)
4 {
5   return valueParam;
6 }
7
8 void myFunction()
9 {
10   int demoObject = 0;
11
12   globalObject = demoObject; // demoObject -> globalObject
13
14   // demoObject -> valueParam; demoObject -> refParam
15   // demoObject -> ptrParam; demoFunction -> result
16   int result = demoFunction(demoObject, demoObject, &demoObject);
17
18   // The following is indirect because it's an expression.
19   // demoObject -> localIndirectObject
20   int localIndirectObject = demoObject + result;
21 }
```

Assigned To - Classic

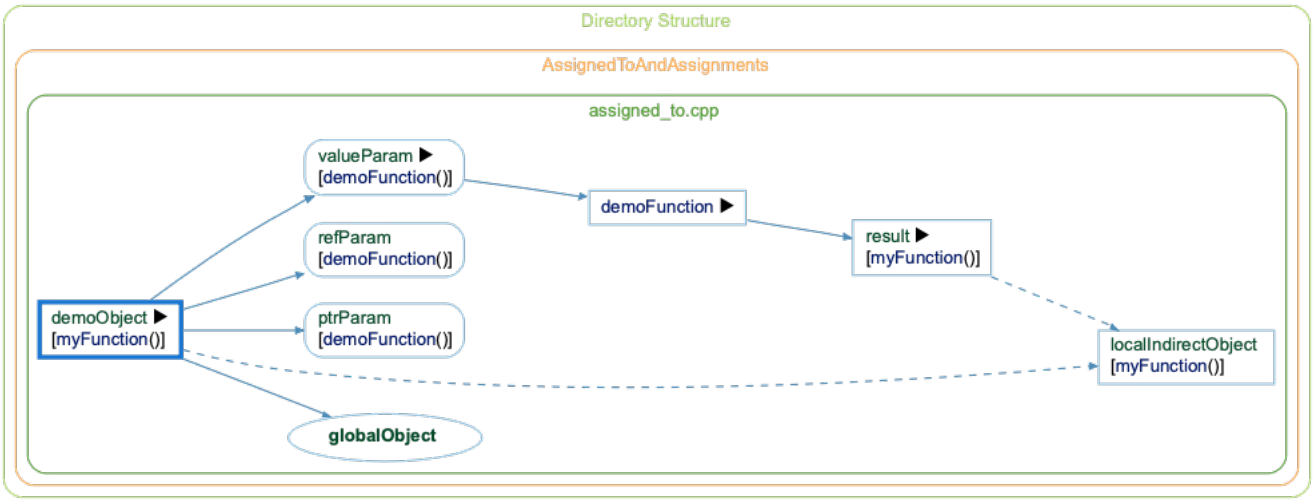
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Assigned To

Assigned To - Cluster

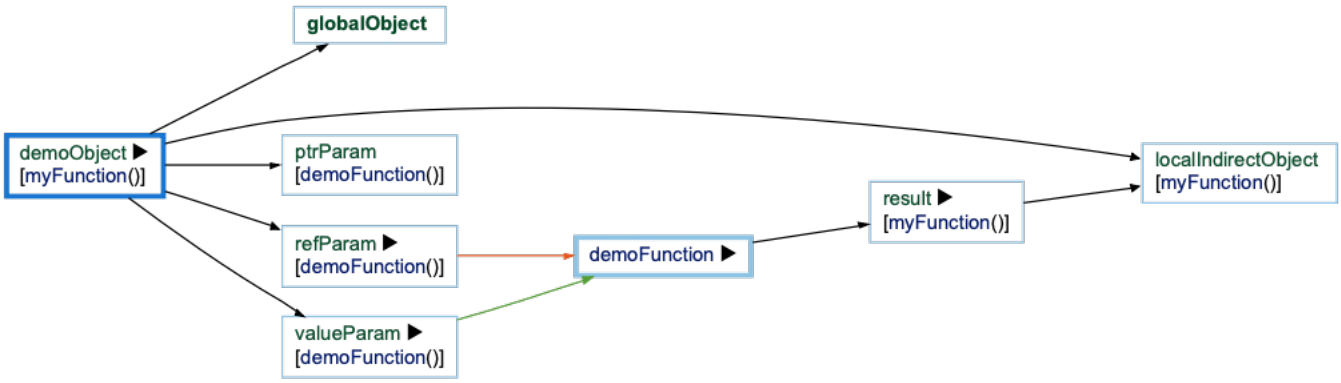
The Cluster variant groups entities by their containing class/file/architecture.



Back to Assigned To

Assigned To - Compare

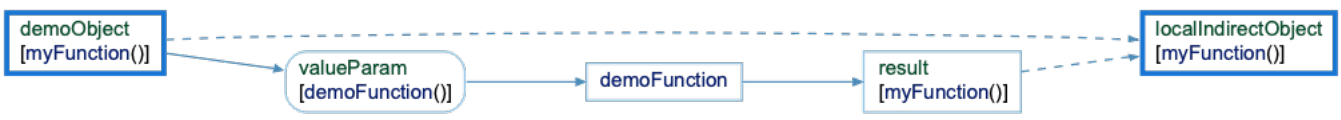
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Assigned To

Assigned To - Relationship

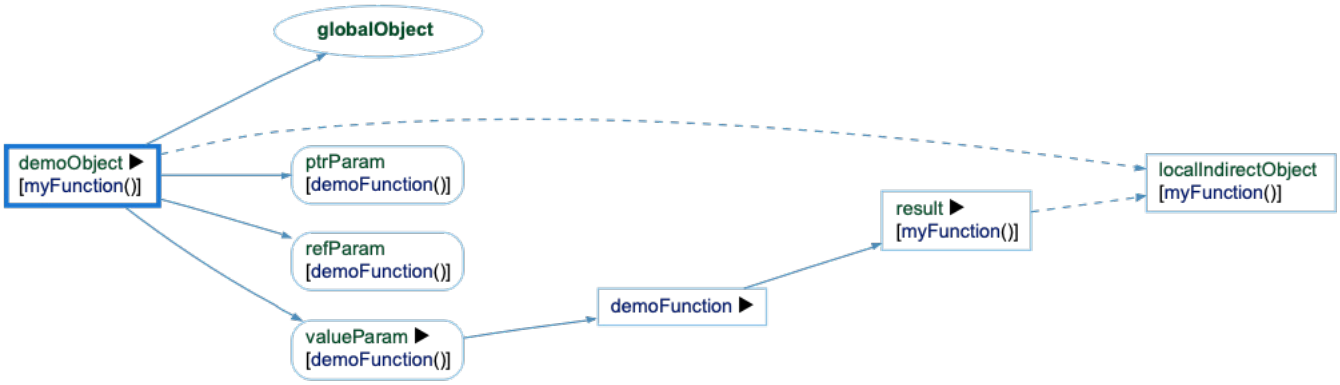
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Assigned To

Assigned To - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Assigned To

Assigned To & Assignments

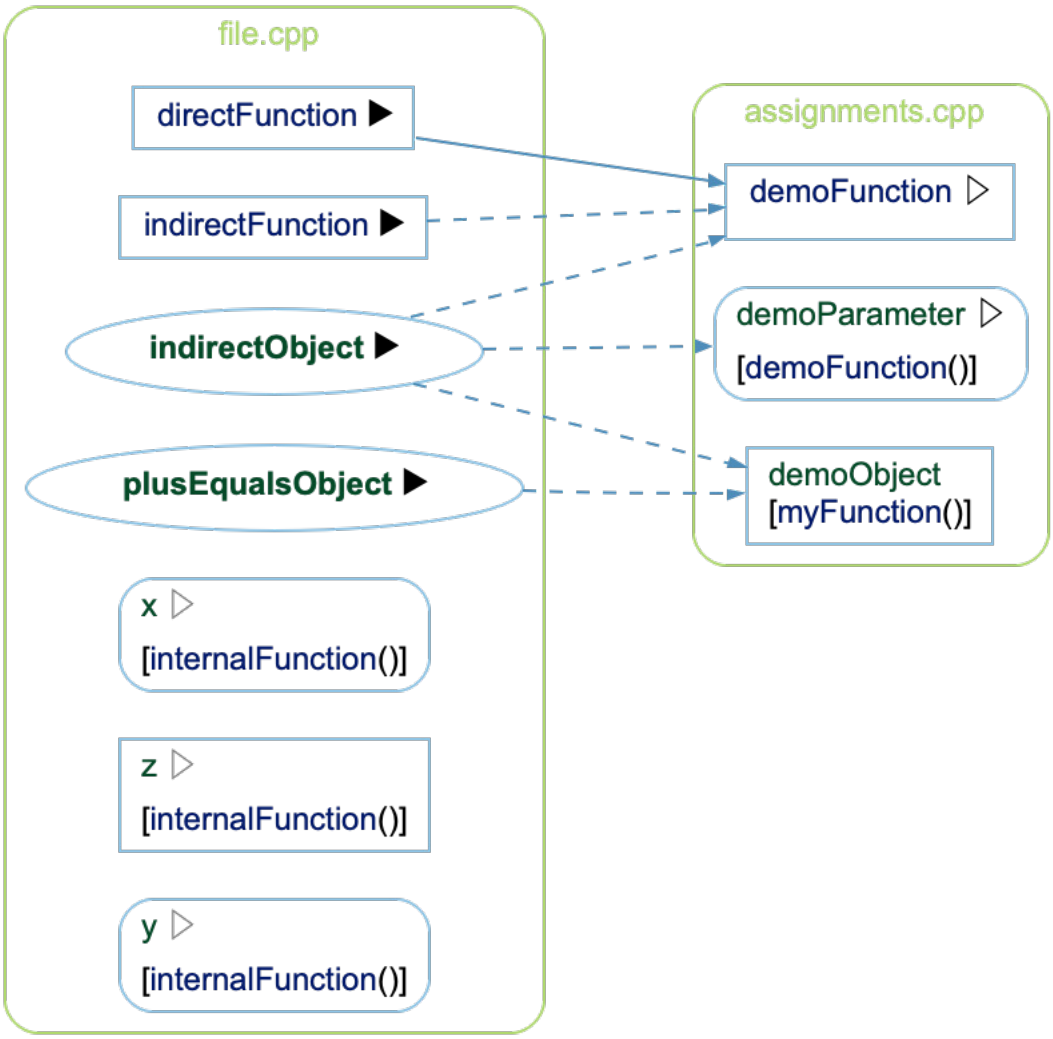
Languages: Ada, C#, C++, Java
Targets: Architectures, Classes, Files
Variants: Assigned To, Assignments, Butterfly, Internal

Show Assigned To and Assignments relationships for a class, file, or architecture

See Assigned To and Assignments graphs for details

Assigned To & Assignments - Assigned To

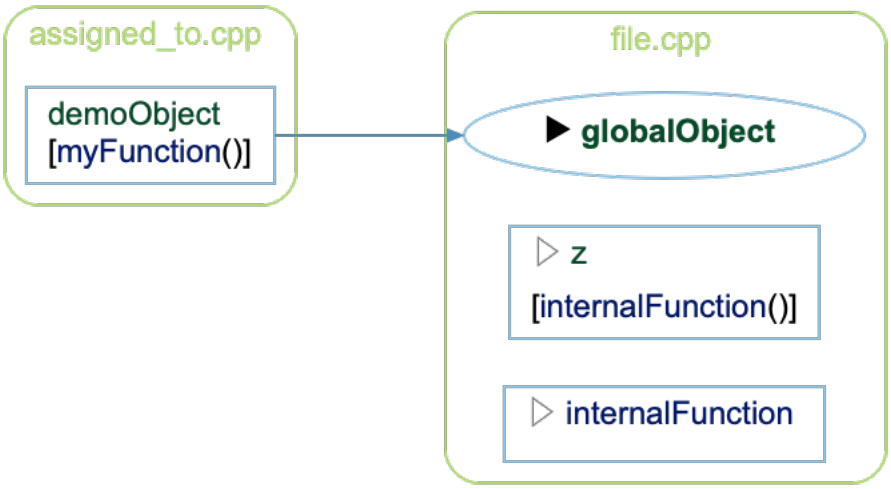
Show Assigned To relationships from entities in a class, file, or architecture



Back to Assigned To & Assignments

Assigned To & Assignments - Assignments

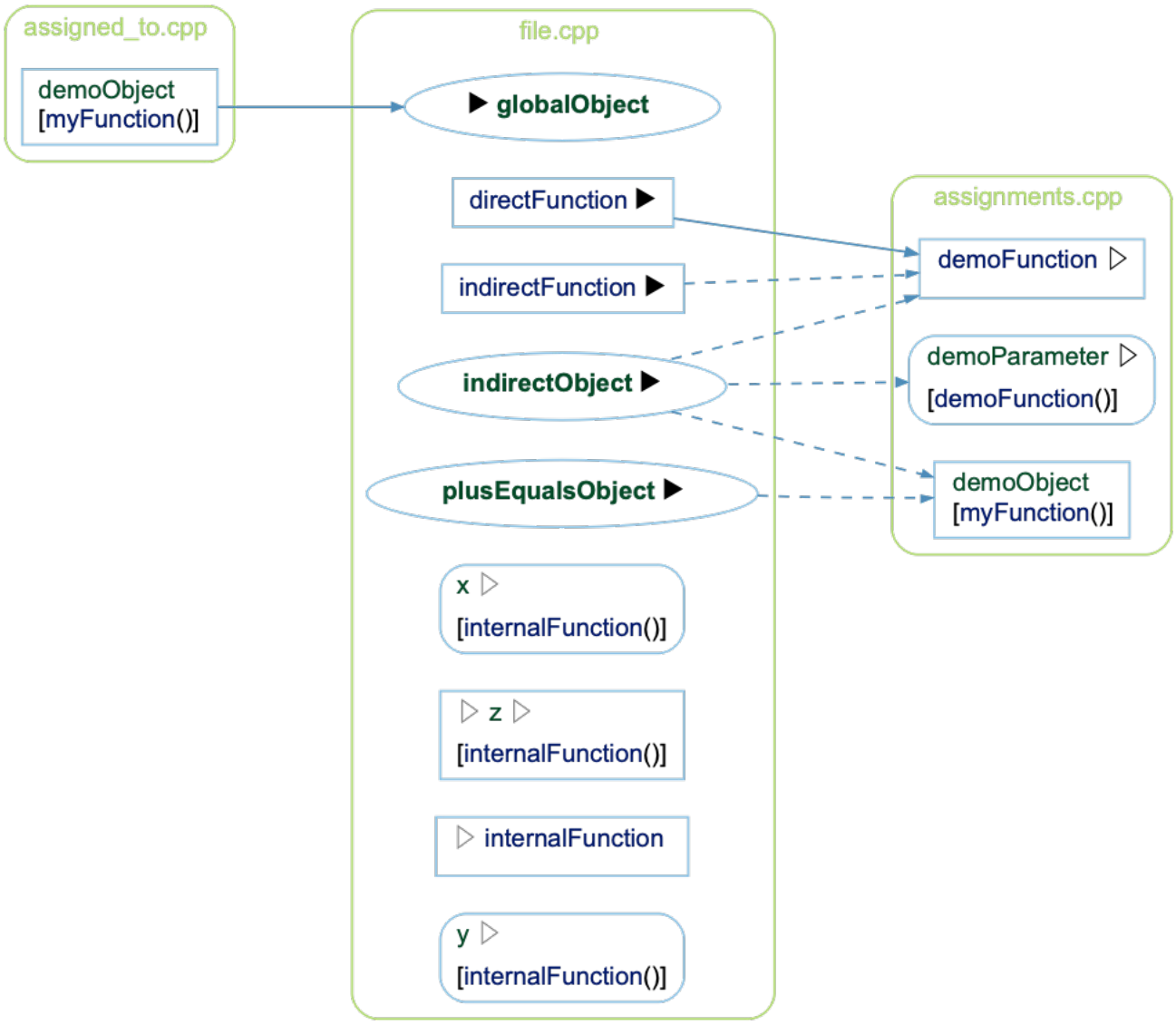
Show Assignments relationships to entities in a class, file, or architecture



Back to Assigned To & Assignments

Assigned To & Assignments - Butterfly

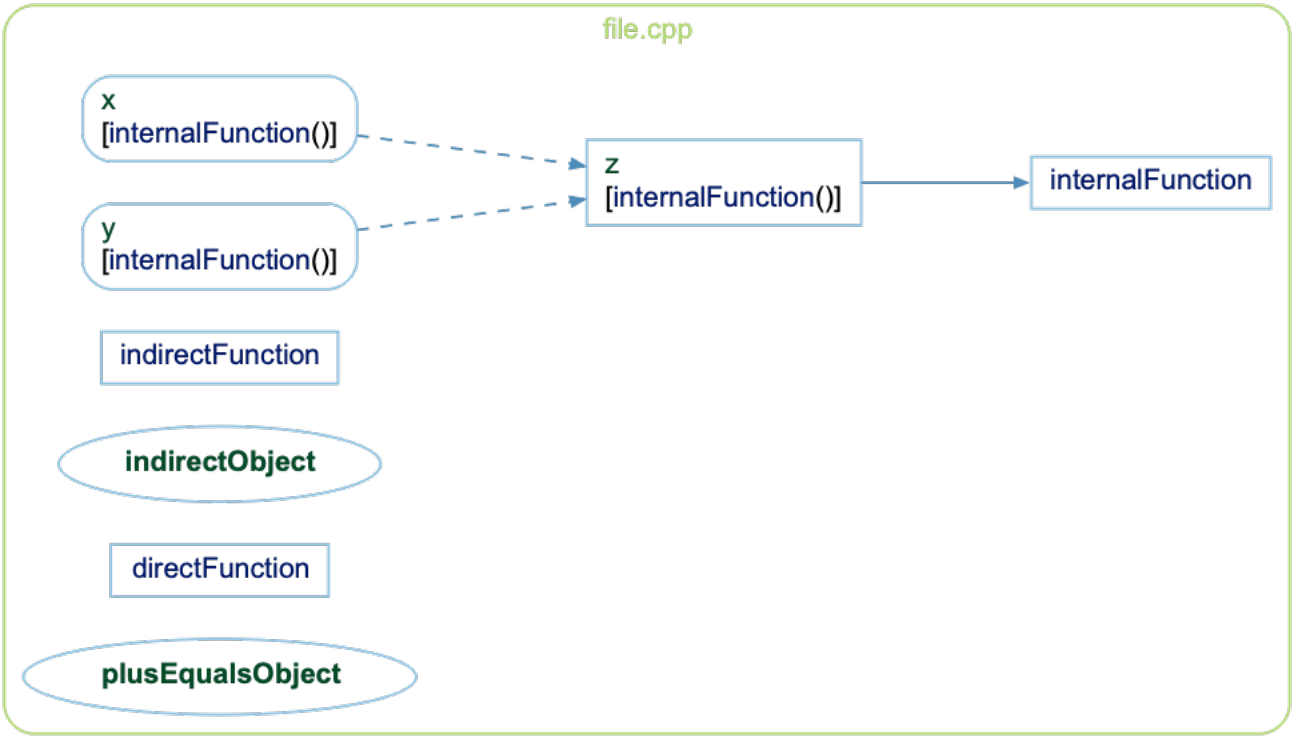
Show both Assigned To relationships from and Assignments relationships to entities in a class, file, or architecture



Back to Assigned To & Assignments

Assigned To & Assignments - Internal

Show Assigned To relationships between entities in a class, file, or architecture



Back to Assigned To & Assignments

Assignments

Languages: Ada, C#, C++, Java
Targets: Functions, Objects
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the objects, functions, and parameters that a selected entity's value is directly or indirectly assigned from.

For a selected object, variable, or parameter, this shows edges from the entities on the right-hand side of assignments where the selected entity is the target. Passing a value as a function argument counts as an assignment to the corresponding parameter. For a selected function, the graph tracks its return expression the same way, treating return statements as assignments to the function itself.

Edges are solid when the source is assigned directly (e.g. demoObject = x;) and dashed when the source is only part of a larger expression (e.g. demoObject = indirectObject + 1;). Compound assignment operators like += and ++ are tracked the same way — since they combine the existing value with another operand, they always produce a dashed (indirect) edge from that operand, e.g. demoObject += plusEqualsObject;. Literal values are never tracked, so an increment like demoObject++; produces no edge at all; use the Variable Tracker graph if you need

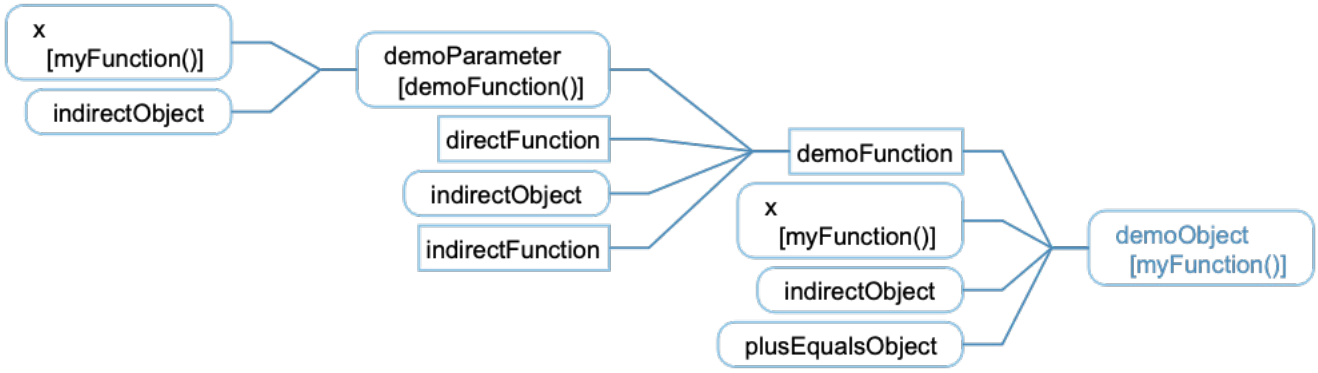
those too. The inverse graph, Assigned To, shows the reverse relationship — the assignments a selected entity's value feeds into.

For example, see the following code and corresponding graph

```
1 extern int directFunction();
2 extern int indirectFunction();
3 extern int indirectObject;
4 extern int plusEqualsObject;
5
6 int demoFunction(int demoParameter)
7 {
8   switch (demoParameter) {
9     case 0: return demoParameter; // demoParameter -> demoFunction()
10     case 1: return directFunction(); // -> demoFunction()
11     case 2: return 0; // literal, no reference
12     default:
13       // All indirect: part of an expression
14       return indirectObject + // indirectObject -> demoFunction()
15              indirectFunction() + // -> demoFunction()
16              1; // literal, no reference
17   }
18 }
19
20 void myFunction(int x)
21 {
22   // demoFunction() -> demoObject; also x -> demoParameter
23   int demoObject = demoFunction(x);
24
25   demoObject = x; // x -> demoObject
26   demoObject = 0; // literal, no reference
27
28   // All indirect: part of an expression
29   demoObject = indirectObject + // indirectObject -> demoObject
30                // demoFunction() -> demoObject; x -> demoParameter
31                demoFunction(indirectObject + 1) +
32                1; //  literal, no reference
33
34   demoObject ++; // modification by literal, no reference
35
36   // modification by plusEqualsObject, indirect assignment
37   demoObject += plusEqualsObject;
38 }
```

Assignments - Classic

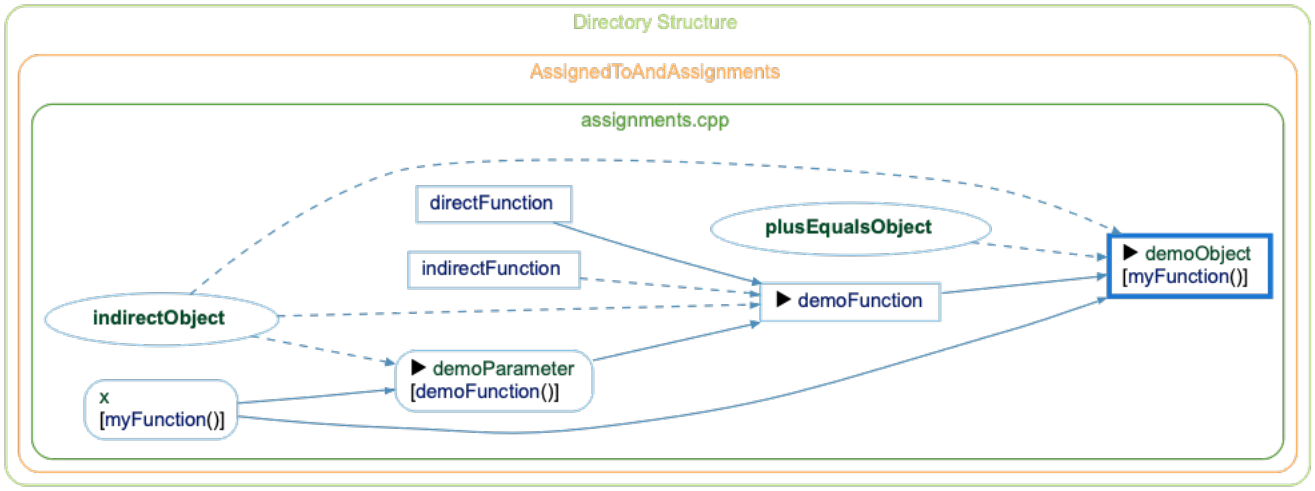
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Assignments

Assignments - Cluster

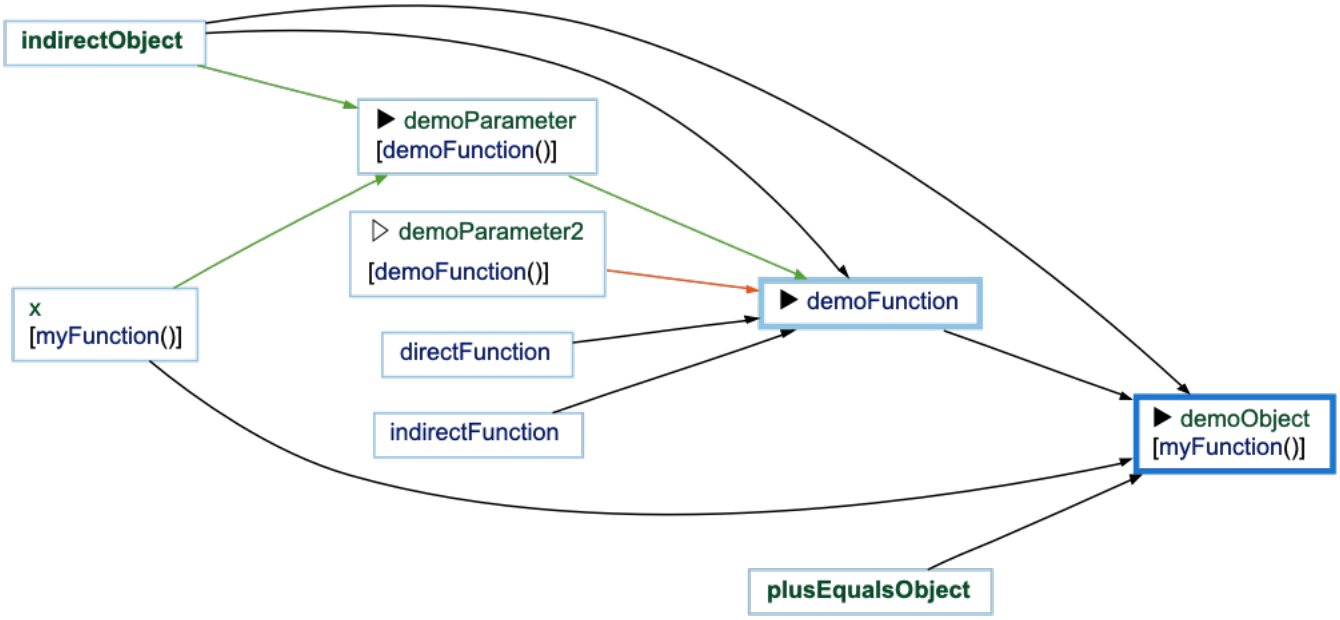
The Cluster variant groups entities by their containing class/file/architecture.



Back to Assignments

Assignments - Compare

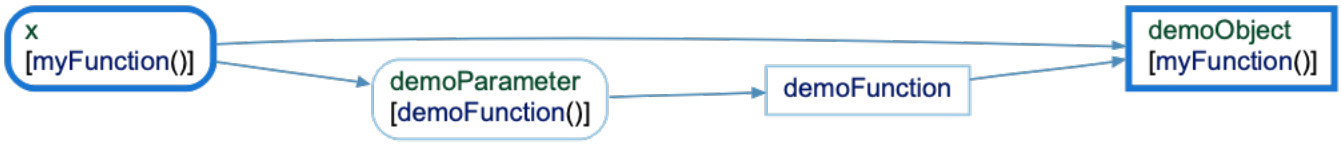
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Assignments

Assignments - Relationship

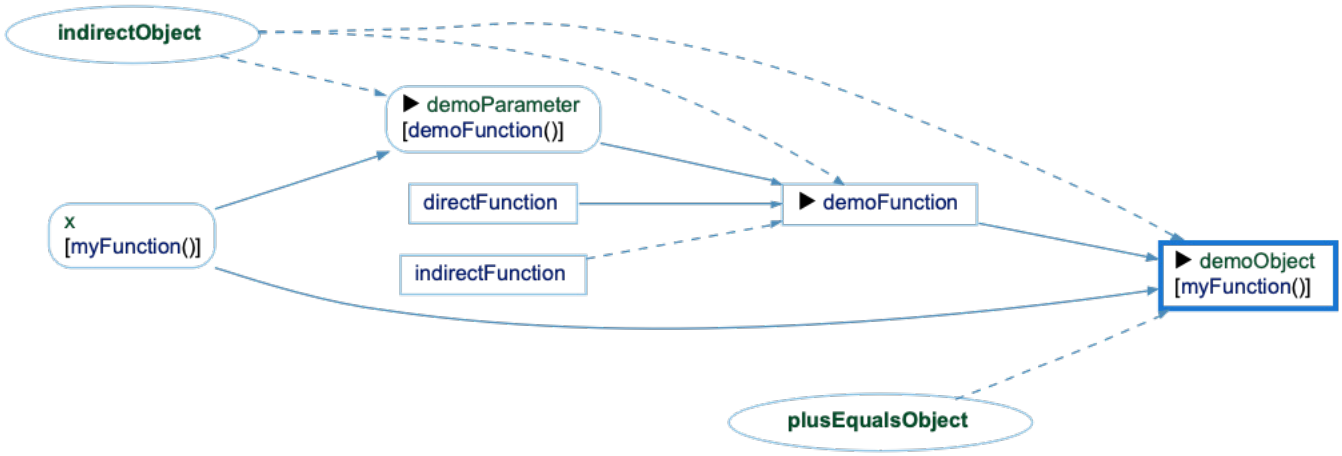
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Assignments

Assignments - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Assignments

Base Classes

Languages: C#, C++, Python, Web
Targets: Classes
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the classes a selected class directly or indirectly inherits from.

For a selected class, struct, or interface, this shows an edge to each class it directly inherits from, then recurses upward into each of those classes' own base classes, building a multi-level ancestry tree. A class with more than one direct base class (multiple inheritance) gets an edge to each one.

Rooted at Vertebrate below, the graph shows edges to its two direct bases, Animal and Eukaryote, then continues from Animal up to LivingThing. The inverse graph, Derived Classes, shows the reverse relationship — everything that inherits from the selected class.

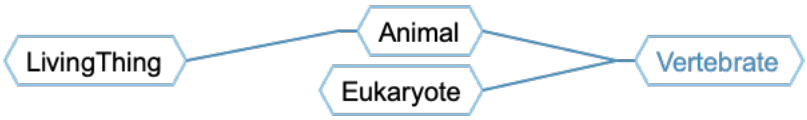
See the following code and corresponding graph

```
1 class LivingThing {
2 public:
3     virtual ~LivingThing() = default;
4     virtual void describe() const;
5 };
6
7 class Animal : public LivingThing {
8 public:
9     void describe() const override;
10 };
11
12 class Eukaryote {
13 public:
14     virtual ~Eukaryote() = default;
15     virtual void classify() const;
16 };
17
18 // Multiple inheritance: Vertebrate has two direct base classes
19 class Vertebrate : public Animal, public Eukaryote {
20 public:
21     void describe() const override;
22     void classify() const override;
23 };
```

Base Classes - Classic

The Classic variant is similar to the Simplified variant, showing all relationships in a

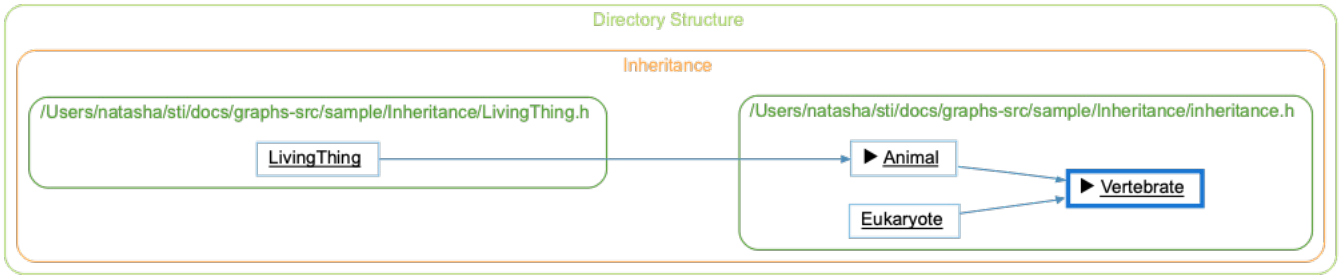
simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Base Classes

Base Classes - Cluster

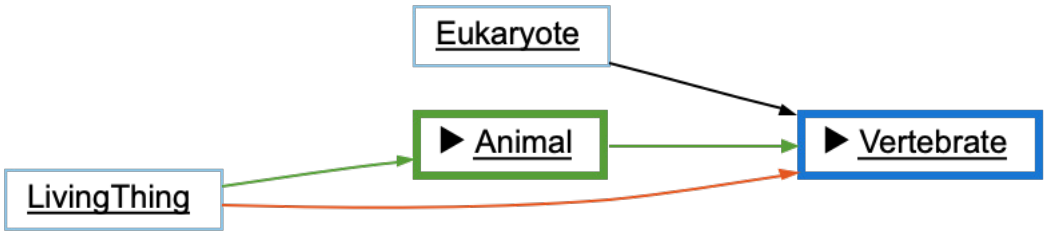
The Cluster variant groups entities by their containing class/file/architecture.



Back to Base Classes

Base Classes - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Base Classes

Base Classes - Relationship

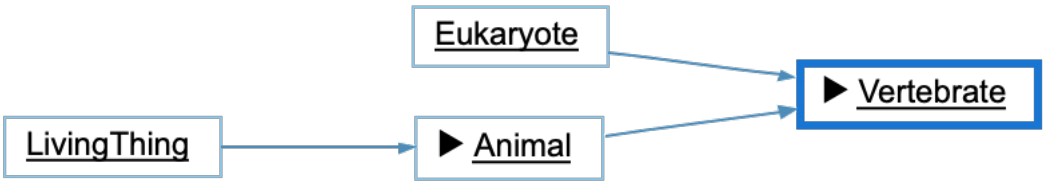
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Base Classes

Base Classes - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Base Classes

Butterfly

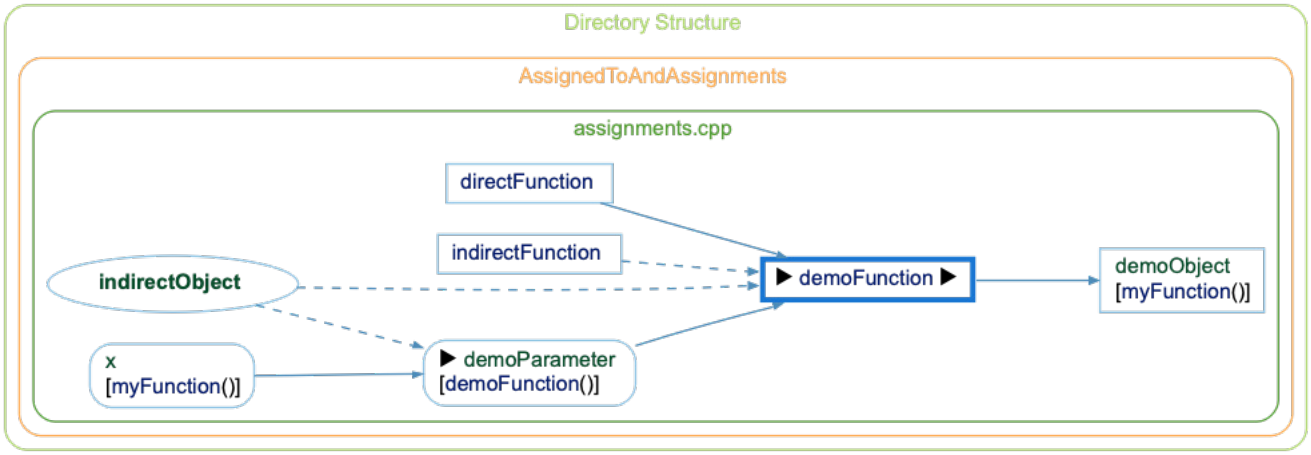
Languages: Ada, Any, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Classes, Files, Functions, Generics, Modules, Objects, Packages, Subprograms, Types
Variants: Assigned To & Assignments (Cluster), Assigned To & Assignments (Compare), Assigned To & Assignments (Simplified), Calls & Called By (Cluster), Calls & Called By (Compare), Calls & Called By (Simplified), Child Lib Units & Parent Lib Unit (Cluster), Child Lib Units & Parent Lib Unit (Compare), Child Lib Units & Parent Lib Unit (Simplified), Classic, Compare (Classic), Data Flow Out & Data Flow In (Cluster), Data Flow Out & Data Flow In (Compare), Data Flow Out & Data Flow In (Simplified), Declaration Tree & Declared In (Cluster), Declaration Tree & Declared In (Compare), Declaration Tree & Declared In (Simplified), Dependencies, Dependencies (Compare), Derived Classes & Base Classes (Cluster), Derived Classes & Base Classes (Compare), Derived Classes & Base Classes (Simplified), Extended By & Extends (Cluster), Extended By & Extends (Compare), Extended By & Extends (Simplified), Imports & Imported By (Cluster), Imports & Imported By (Compare), Imports & Imported By (Simplified), Include & Includeby (Cluster), Include & Includeby (Compare), Include & Includeby (Simplified), Overriddenby & Overrides (Cluster), Overriddenby & Overrides (Compare), Overriddenby & Overrides (Simplified), Requires & Required By (Cluster), Requires & Required By (Compare), Requires & Required By (Simplified), Type Tree & Type Derived From (Cluster), Type Tree & Type Derived From (Compare), Type Tree & Type Derived From (Simplified), Uses & Used By (Cluster), Uses & Used By (Compare), Uses & Used By (Simplified), Uses Traits & Used By (Cluster), Uses Traits & Used By (Compare), Uses Traits & Used By (Simplified), With & Withby (Cluster), With & Withby (Compare), With & Withby (Simplified)

Butterfly - Assigned To & Assignments (Cluster)

Show both Assigned To and Assignments relationships

See Assigned To and Assignments graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



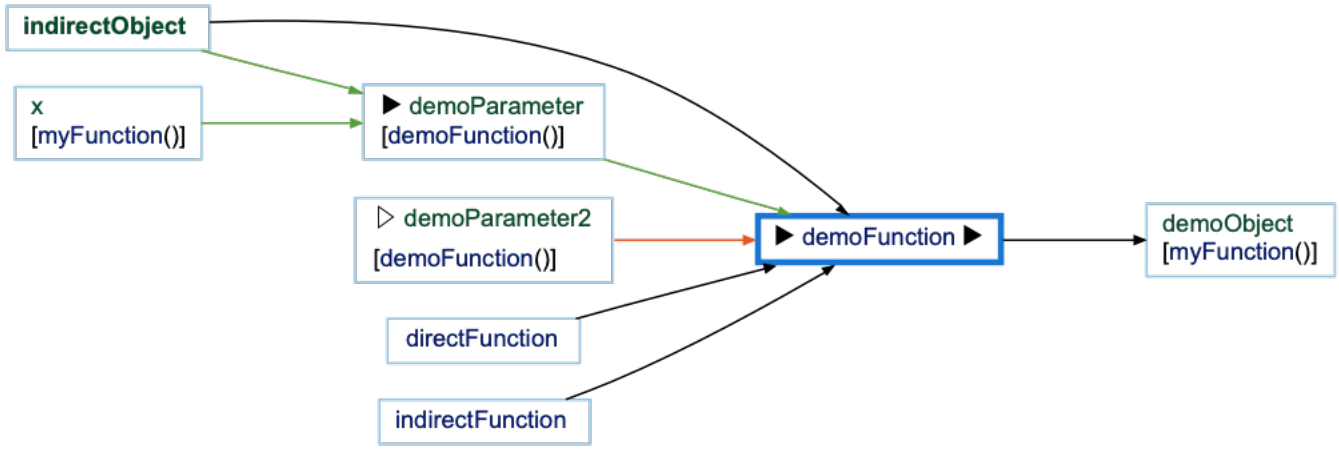
Back to Butterfly

Butterfly - Assigned To & Assignments (Compare)

Show both Assigned To and Assignments relationships

See Assigned To and Assignments graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



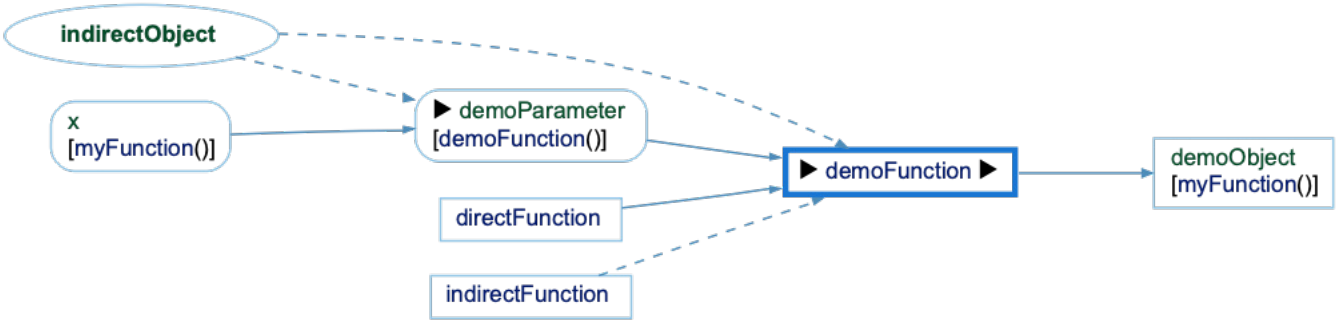
Back to Butterfly

Butterfly - Assigned To & Assignments (Simplified)

Show both Assigned To and Assignments relationships

See Assigned To and Assignments graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



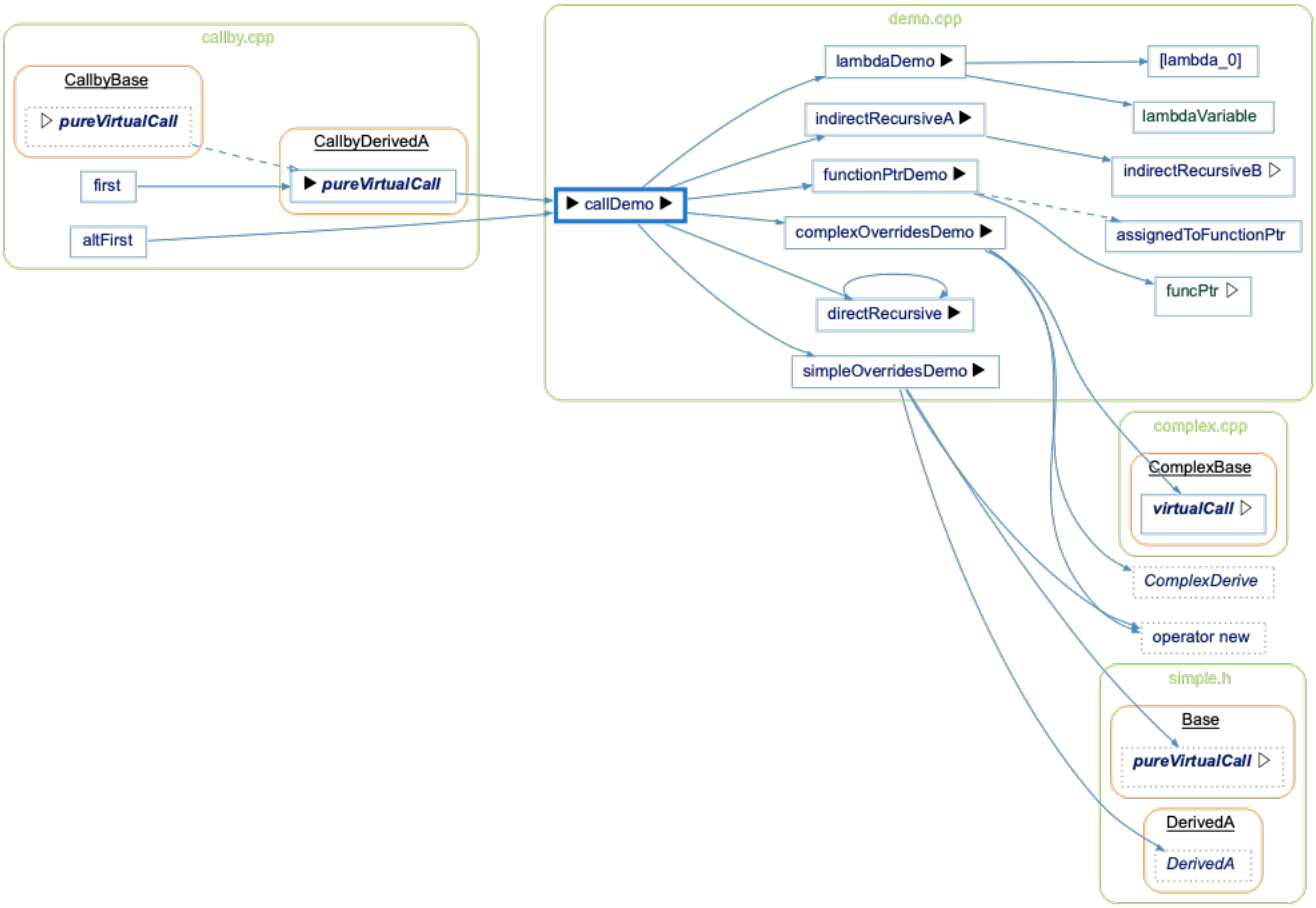
Back to Butterfly

Butterfly - Calls & Called By (Cluster)

Show both Calls and Called By relationships

See Calls and Called By graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



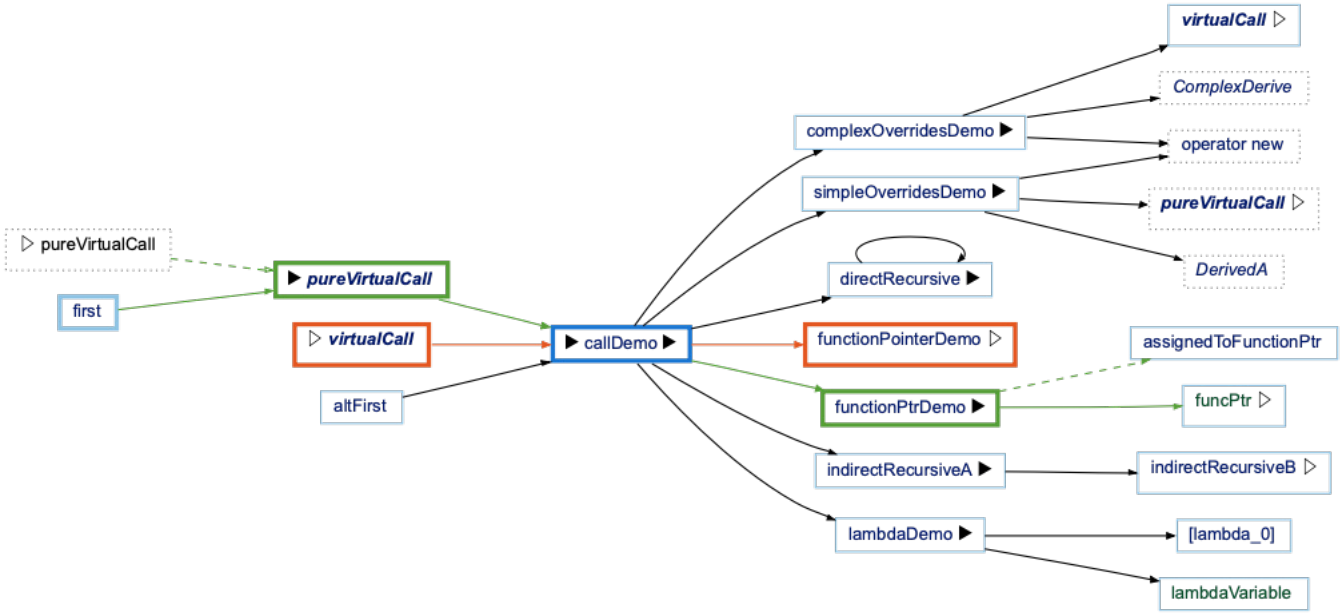
Back to Butterfly

Butterfly - Calls & Called By (Compare)

Show both Calls and Called By relationships

See Calls and Called By graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



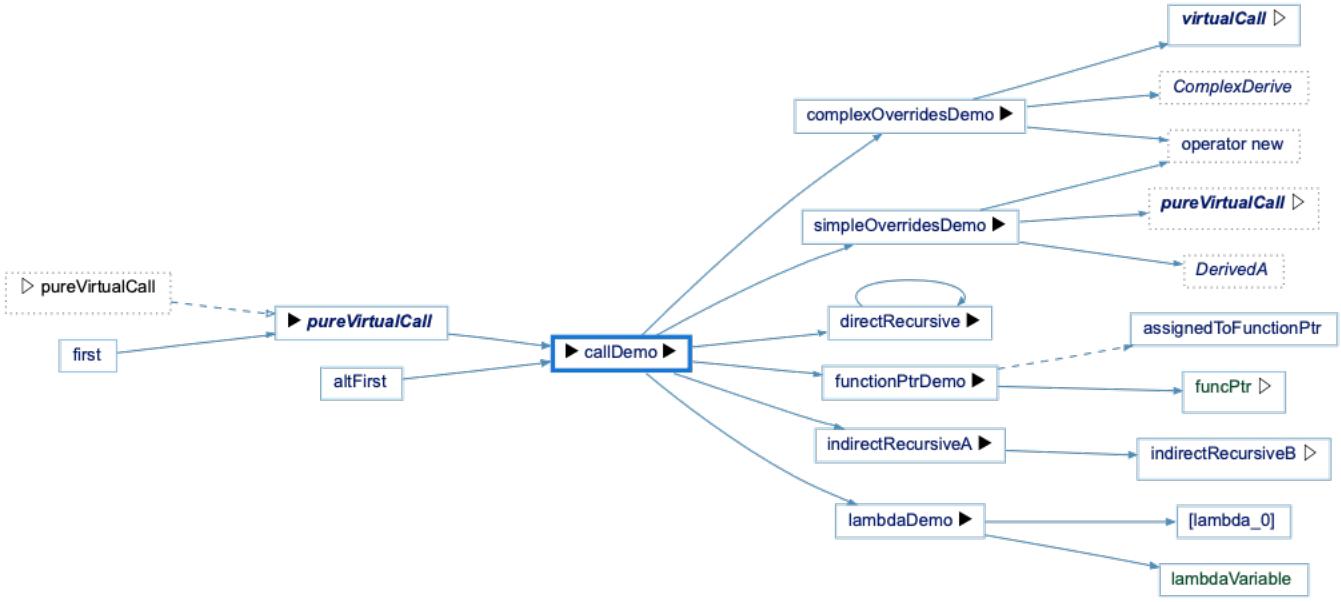
Back to Butterfly

Butterfly - Calls & Called By (Simplified)

Show both Calls and Called By relationships

See Calls and Called By graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



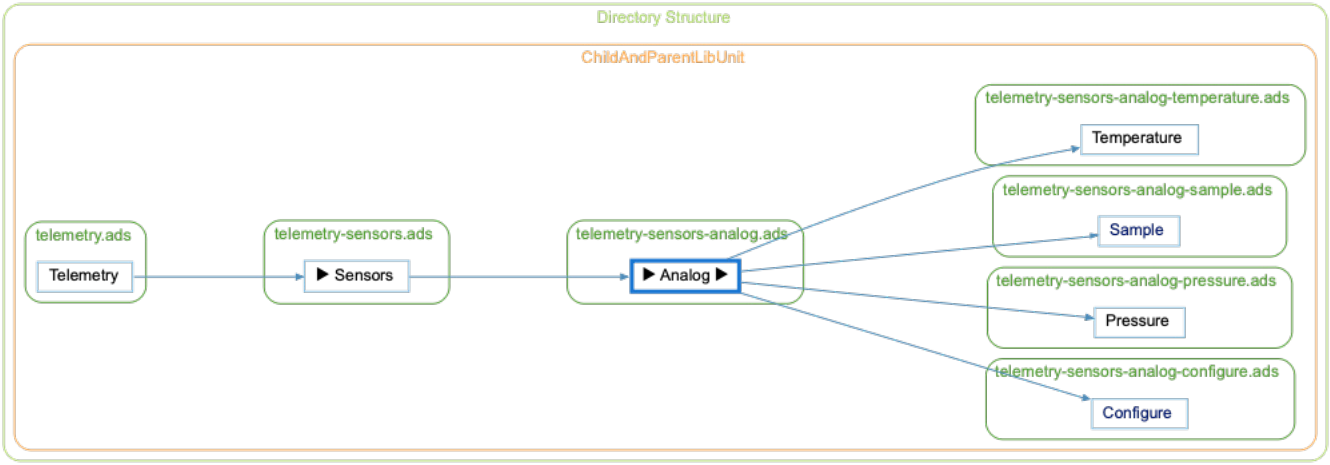
Back to Butterfly

Butterfly - Child Lib Units & Parent Lib Unit (Cluster)

Show both Child Lib Units and Parent Lib Unit relationships

See Child Lib Units and Parent Lib Unit graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



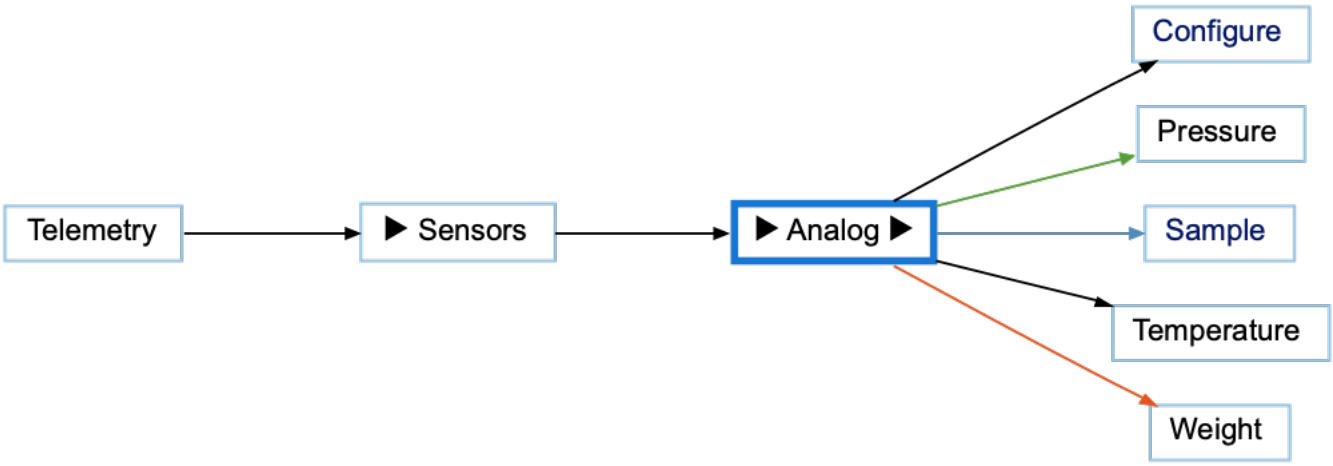
Back to Butterfly

Butterfly - Child Lib Units & Parent Lib Unit (Compare)

Show both Child Lib Units and Parent Lib Unit relationships

See Child Lib Units and Parent Lib Unit graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



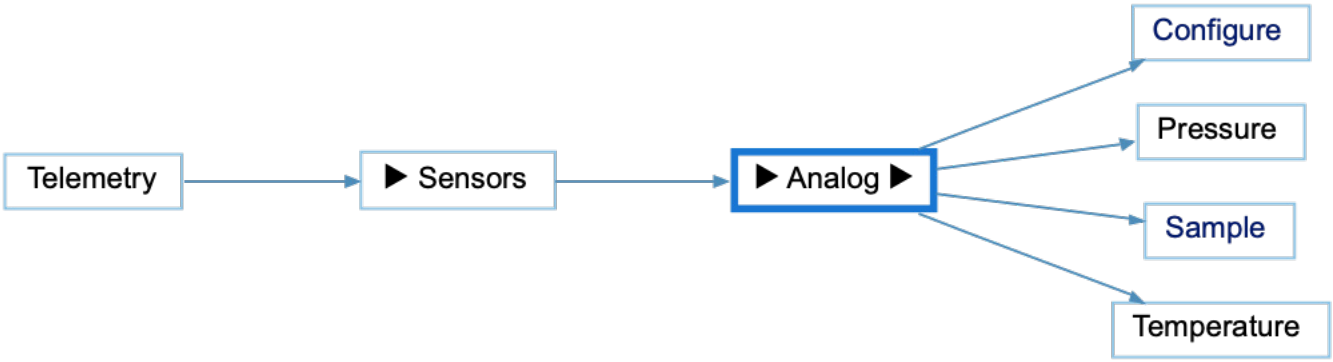
Back to Butterfly

Butterfly - Child Lib Units & Parent Lib Unit (Simplified)

Show both Child Lib Units and Parent Lib Unit relationships

See Child Lib Units and Parent Lib Unit graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



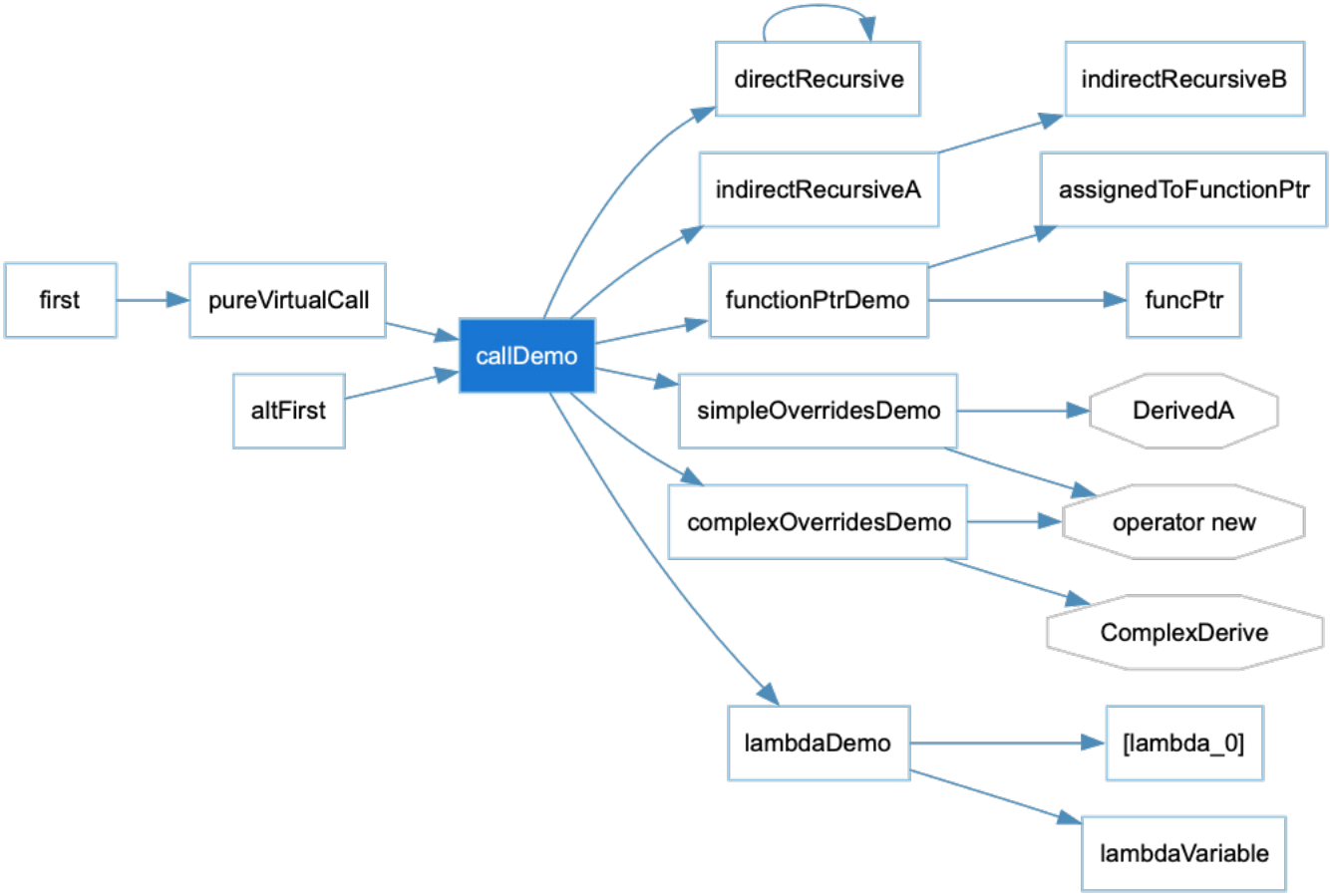
Back to Butterfly

Butterfly - Classic

See a selected entity's full relationship neighborhood in one diagram: everything it depends on, and everything that depends on it, at once.

Which relationship is shown depends on what's selected: functions and subprograms show Calls and Called By; classes, structs, and interfaces show Base Classes and Derived Classes (or Extends and Extended By for Java and Pascal); Ada types show their derivation chain; files show Include and Includeby; VHDL entities show Use and Useby. That makes it useful for impact analysis — seeing at a glance what a change to the selected entity could affect both upstream and downstream — without opening two separate graphs.

See also the compare graph.



This variant is a plugin. It can be enabled/disabled from the Plugin Manager.

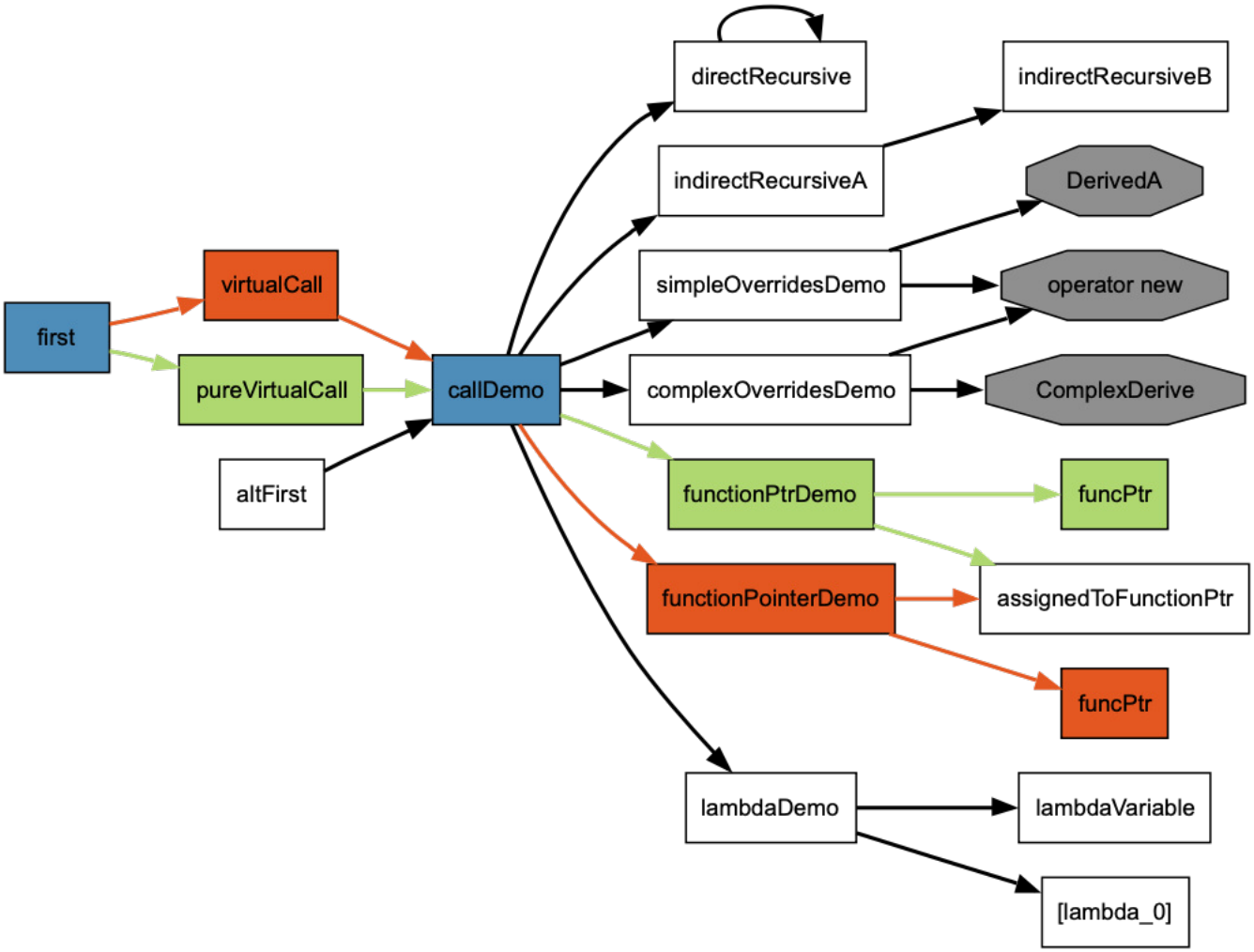
Back to Butterfly

Butterfly - Compare (Classic)

Compare a selected entity's full relationship neighborhood between two versions of the project, in one diagram: everything it depends on, and everything that depends on it, with entities and edges colored by whether they were added, removed, or changed.

Which relationship is compared depends on what's selected, the same as in the Classic variant: functions and subprograms compare Calls and Called By; classes, structs, and interfaces compare Base Classes and Derived Classes; and so on. That makes it useful for reviewing how a function's or class's full calling or inheritance context changed across a refactor or release — whether new callers appeared, old ones disappeared, or an override was added or removed — without diffing two separate graphs.

See also the classic graph.



This variant is a plugin. It can be enabled/disabled from the Plugin Manager.

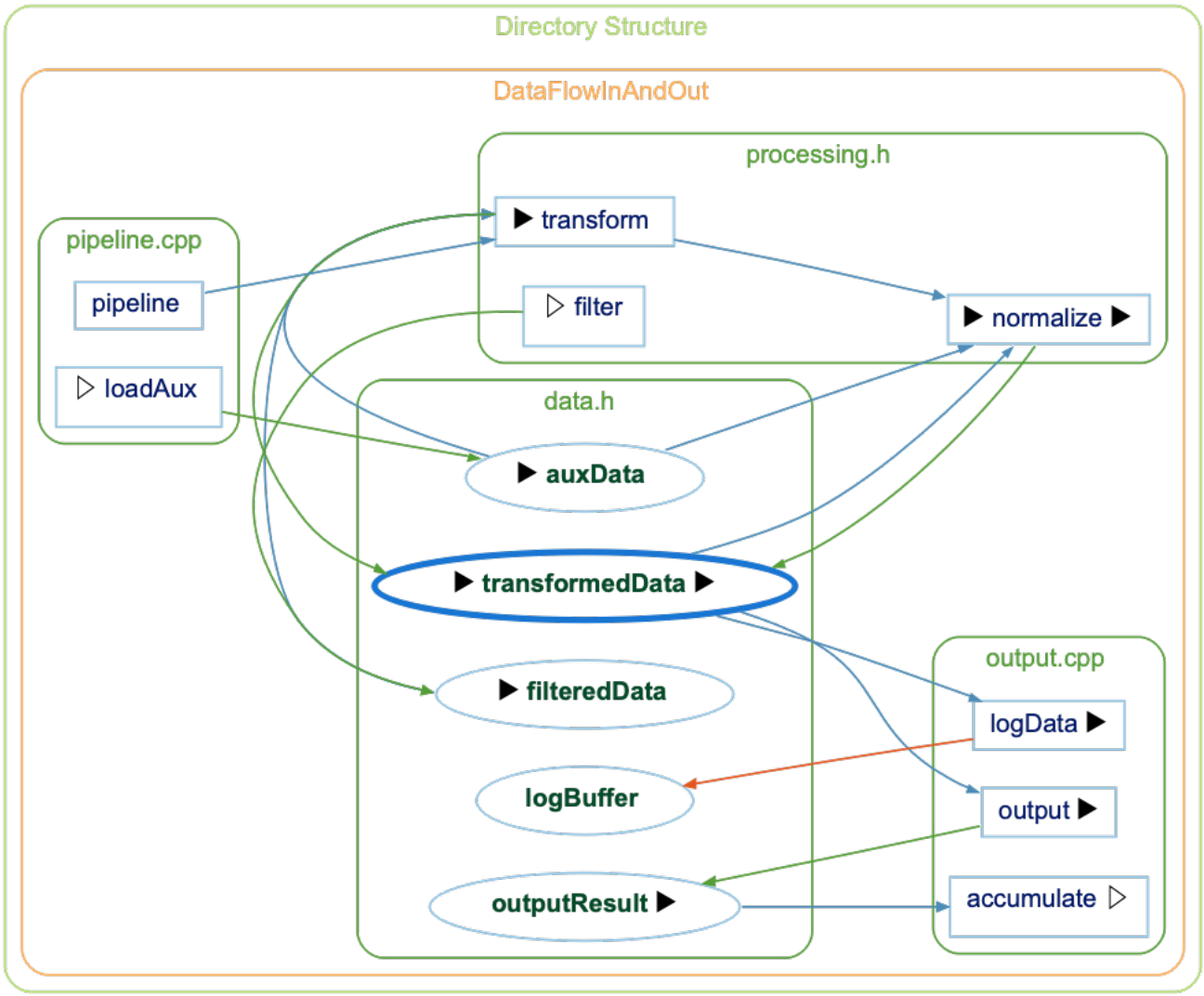
Back to Butterfly

Butterfly - Data Flow Out & Data Flow In (Cluster)

Show both Data Flow Out and Data Flow In relationships

See Data Flow Out and Data Flow In graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



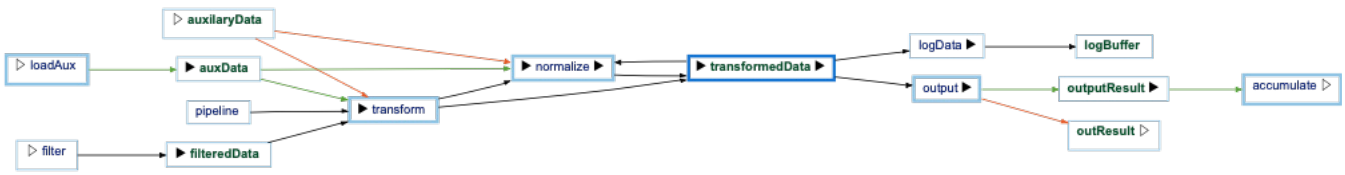
Back to Butterfly

Butterfly - Data Flow Out & Data Flow In (Compare)

Show both Data Flow Out and Data Flow In relationships

See Data Flow Out and Data Flow In graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Butterfly

Butterfly - Data Flow Out & Data Flow In (Simplified)

Show both Data Flow Out and Data Flow In relationships

See Data Flow Out and Data Flow In graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



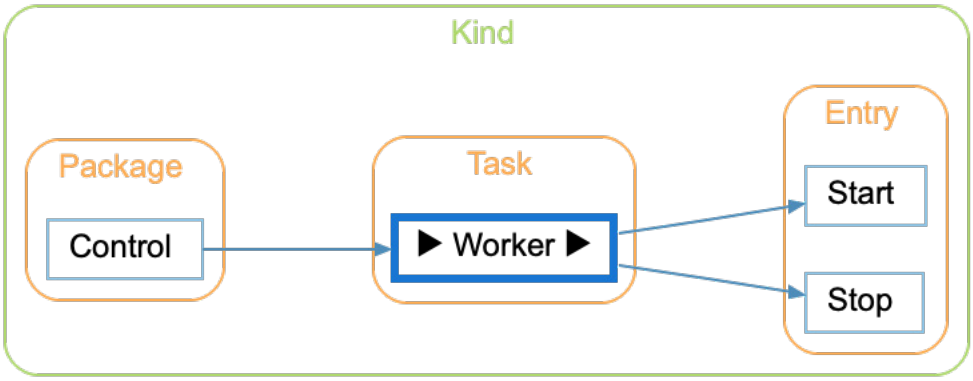
Back to Butterfly

Butterfly - Declaration Tree & Declared In (Cluster)

Show both Declaration Tree and Declared In relationships

See Declaration Tree and Declared In graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



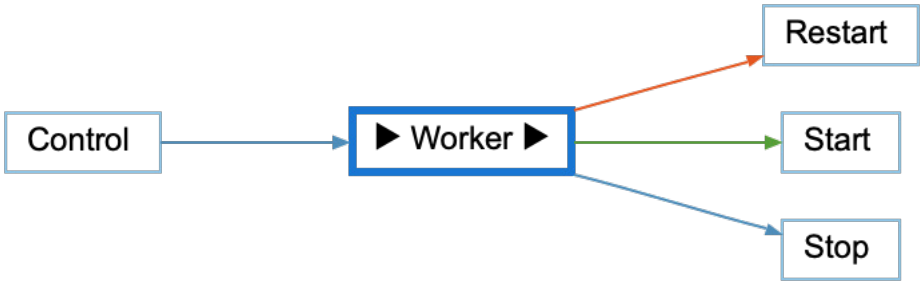
Back to Butterfly

Butterfly - Declaration Tree & Declared In (Compare)

Show both Declaration Tree and Declared In relationships

See Declaration Tree and Declared In graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Butterfly

Butterfly - Declaration Tree & Declared In (Simplified)

Show both Declaration Tree and Declared In relationships

See Declaration Tree and Declared In graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



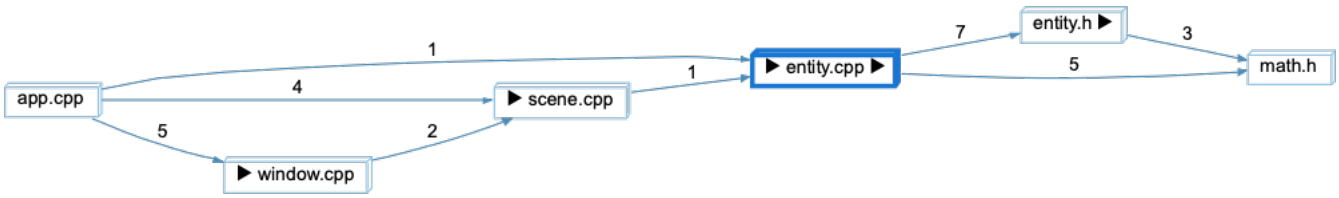
Back to Butterfly

Butterfly - Dependencies

Show both Depends On and Depended On By relationships.

See Depends On and Depended On By graphs for details.

This variant shows each file/class once. In the GUI, the files/classes can be expanded to show the entities for each reference.



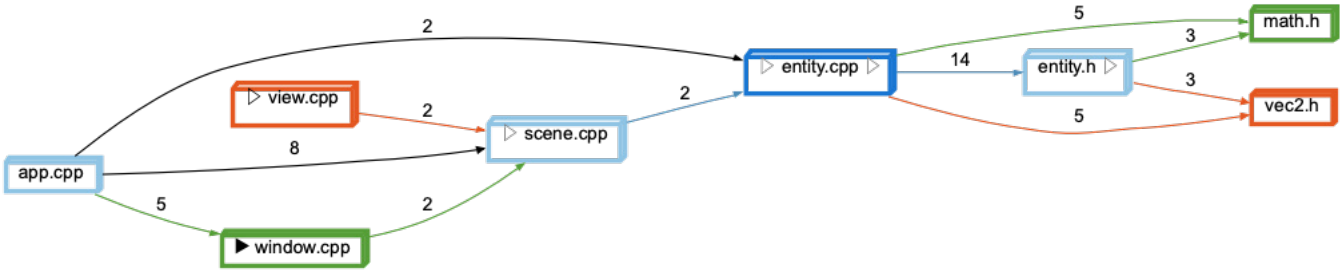
Back to Butterfly

Butterfly - Dependencies (Compare)

Show both Depends On and Depended On By relationships.

See Depends On and Depended On By graphs for details.

The Compare variant shows how the graph has changed relative to the project's comparison database.



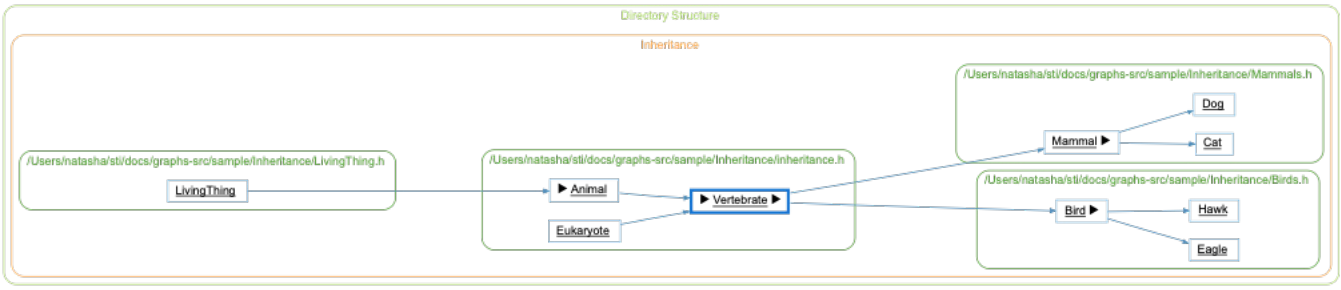
Back to Butterfly

Butterfly - Derived Classes & Base Classes (Cluster)

Show both Derived Classes and Base Classes relationships

See Derived Classes and Base Classes graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



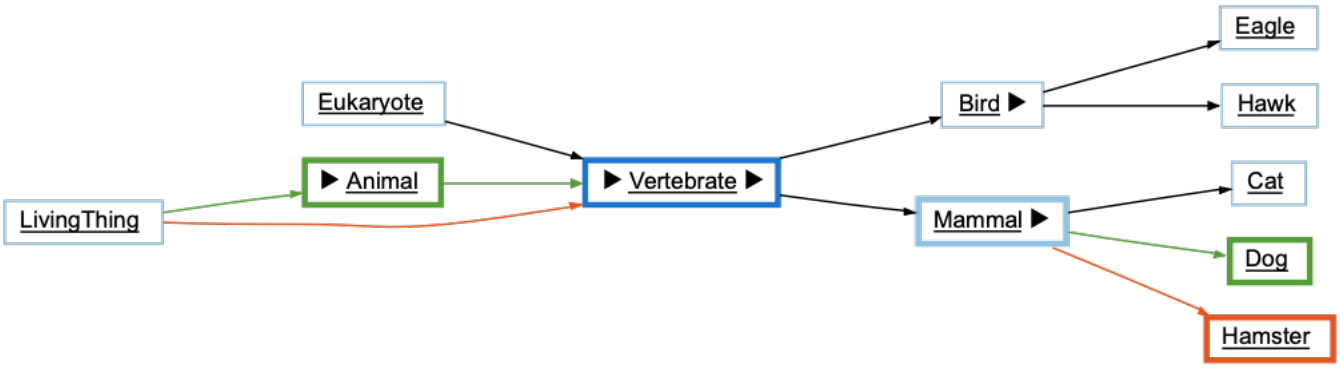
Back to Butterfly

Butterfly - Derived Classes & Base Classes (Compare)

Show both Derived Classes and Base Classes relationships

See Derived Classes and Base Classes graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



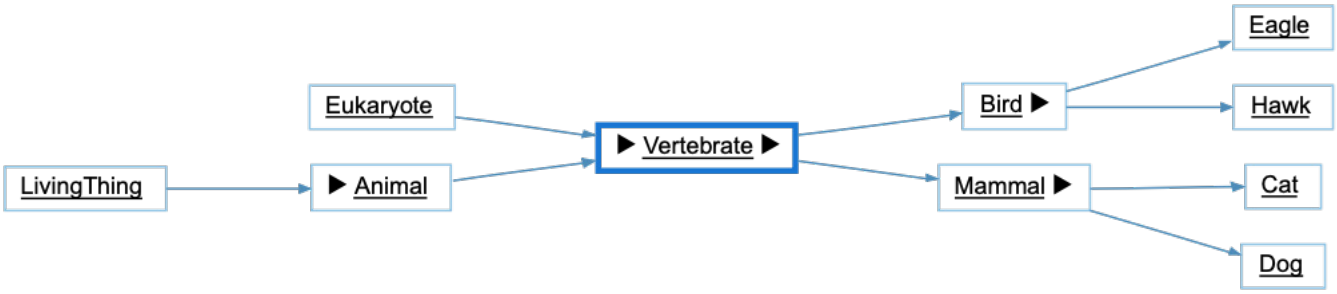
Back to Butterfly

Butterfly - Derived Classes & Base Classes (Simplified)

Show both Derived Classes and Base Classes relationships

See Derived Classes and Base Classes graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



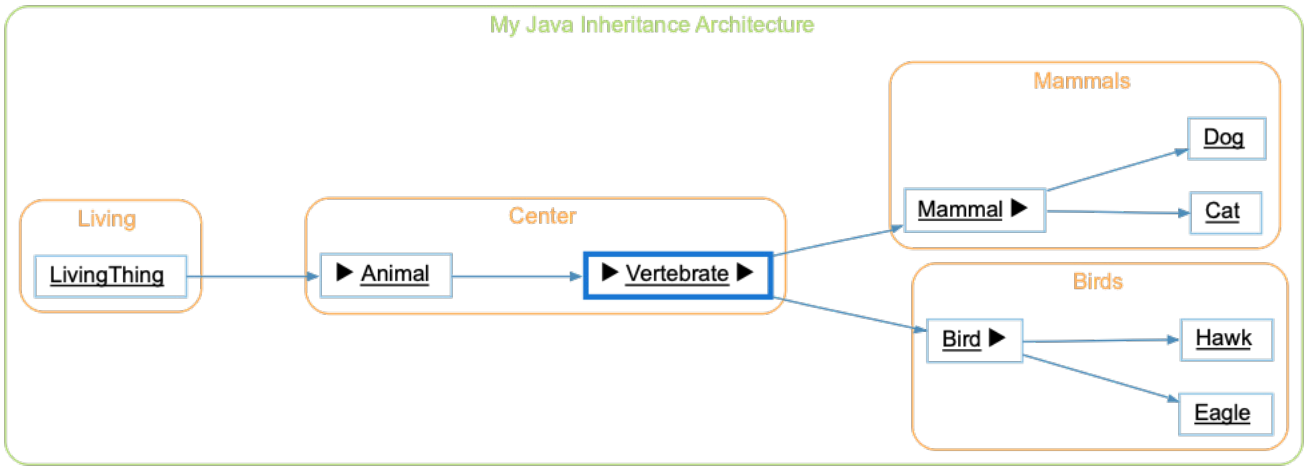
Back to Butterfly

Butterfly - Extended By & Extends (Cluster)

Show both Extended By and Extends relationships

See Extended By and Extends graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



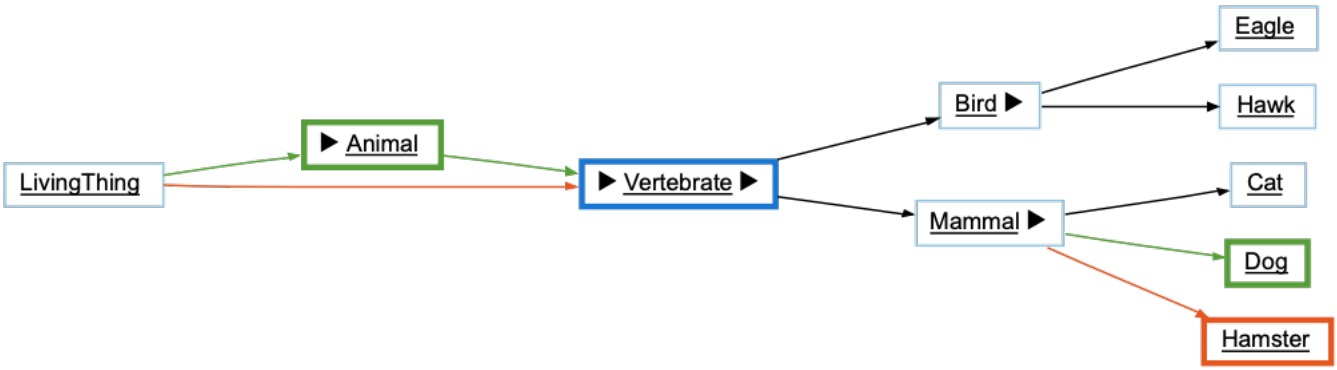
Back to Butterfly

Butterfly - Extended By & Extends (Compare)

Show both Extended By and Extends relationships

See Extended By and Extends graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



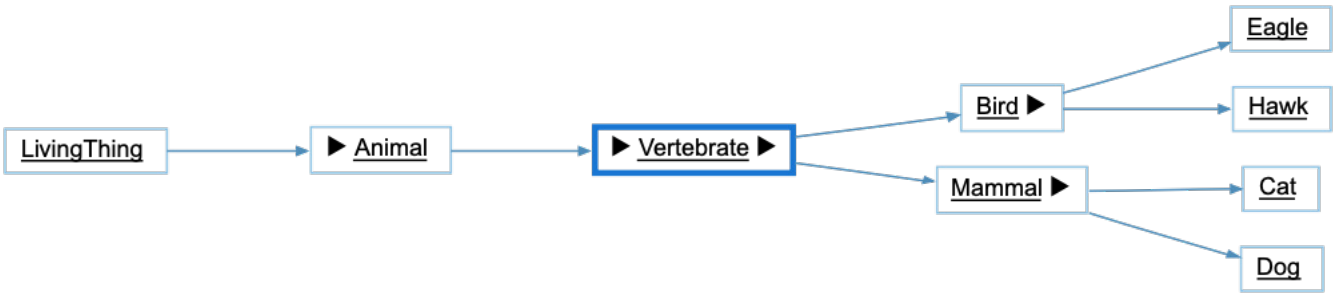
Back to Butterfly

Butterfly - Extended By & Extends (Simplified)

Show both Extended By and Extends relationships

See Extended By and Extends graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



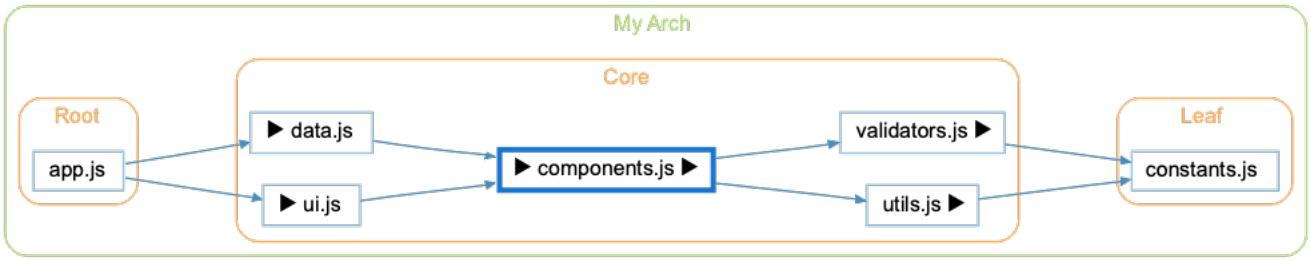
Back to Butterfly

Butterfly - Imports & Imported By (Cluster)

Show both Imports and Imported By relationships

See Imports and Imported By graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



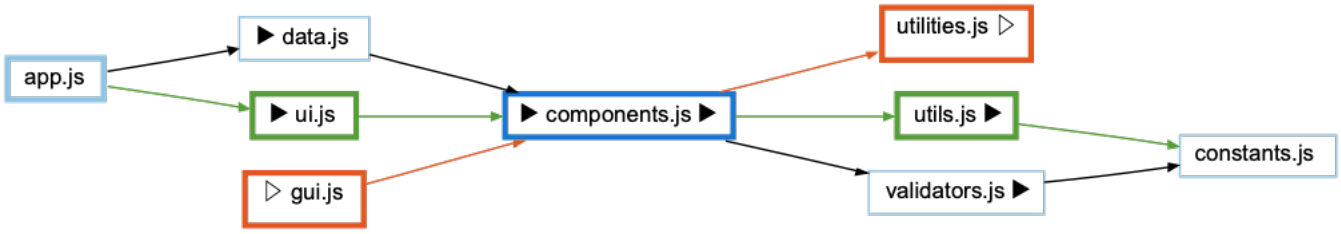
Back to Butterfly

Butterfly - Imports & Imported By (Compare)

Show both Imports and Imported By relationships

See Imports and Imported By graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Butterfly

Butterfly - Imports & Imported By (Simplified)

Show both Imports and Imported By relationships

See Imports and Imported By graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



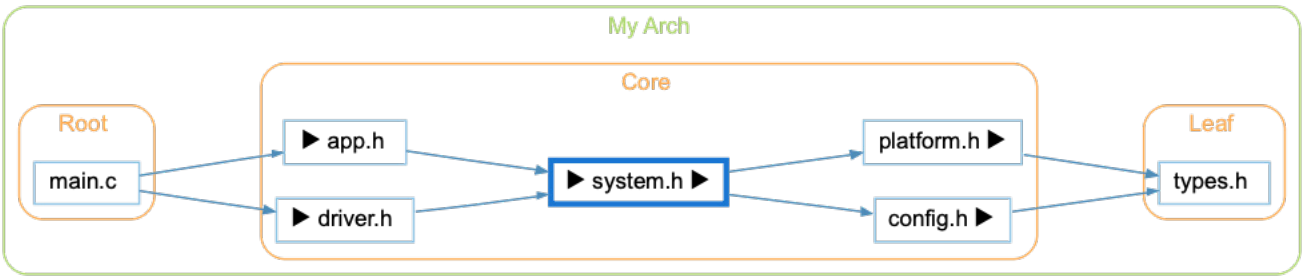
Back to Butterfly

Butterfly - Include & Includeby (Cluster)

Show both Include and Includeby relationships

See Include and Includeby graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



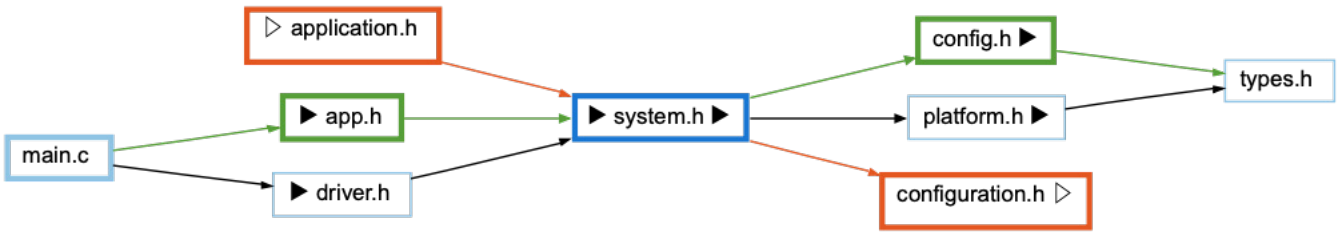
Back to Butterfly

Butterfly - Include & Includeby (Compare)

Show both Include and Includeby relationships

See Include and Includeby graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Butterfly

Butterfly - Include & Includeby (Simplified)

Show both Include and Includeby relationships

See Include and Includeby graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



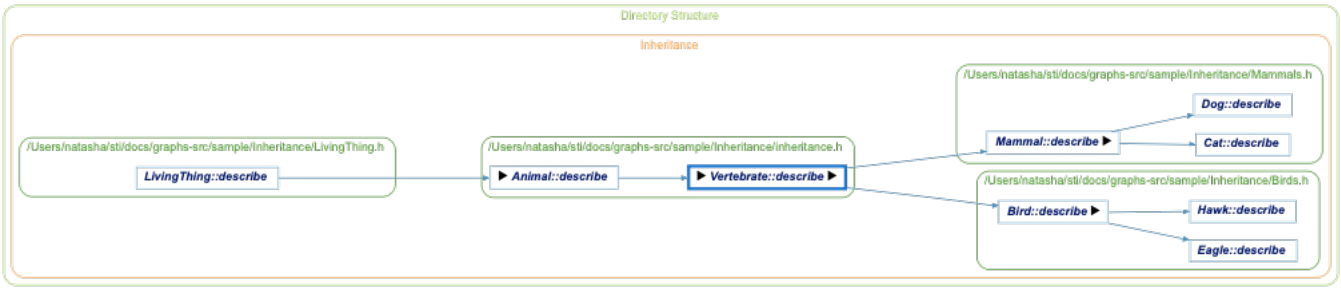
Back to Butterfly

Butterfly - Overriddenby & Overrides (Cluster)

Show both Overriddenby and Overrides relationships

See Overriddenby and Overrides graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



Back to Butterfly

Butterfly - Overriddenby & Overrides (Compare)

Show both Overriddenby and Overrides relationships

See Overriddenby and Overrides graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Butterfly

Butterfly - Overriddenby & Overrides (Simplified)

Show both Overriddenby and Overrides relationships

See Overriddenby and Overrides graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



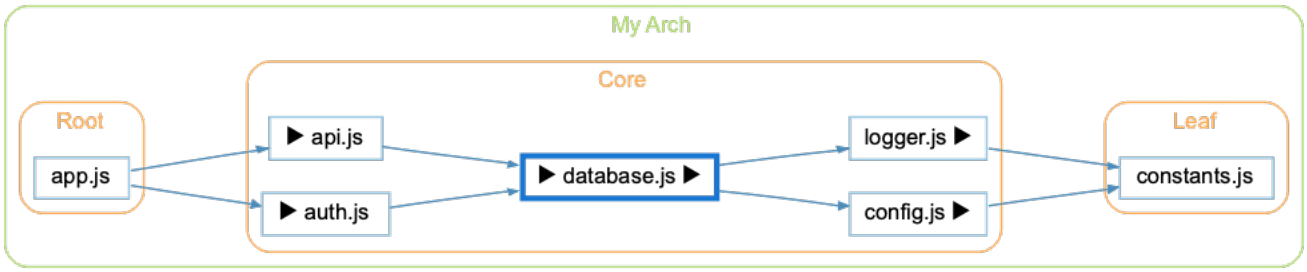
Back to Butterfly

Butterfly - Requires & Required By (Cluster)

Show both Requires and Required By relationships

See Requires and Required By graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



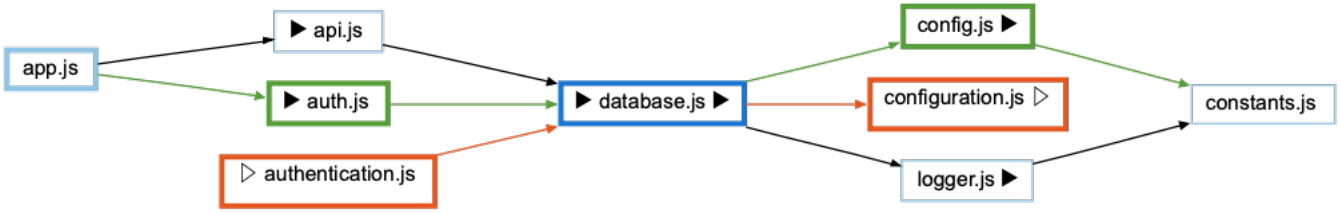
Back to Butterfly

Butterfly - Requires & Required By (Compare)

Show both Requires and Required By relationships

See Requires and Required By graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Butterfly

Butterfly - Requires & Required By (Simplified)

Show both Requires and Required By relationships

See Requires and Required By graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



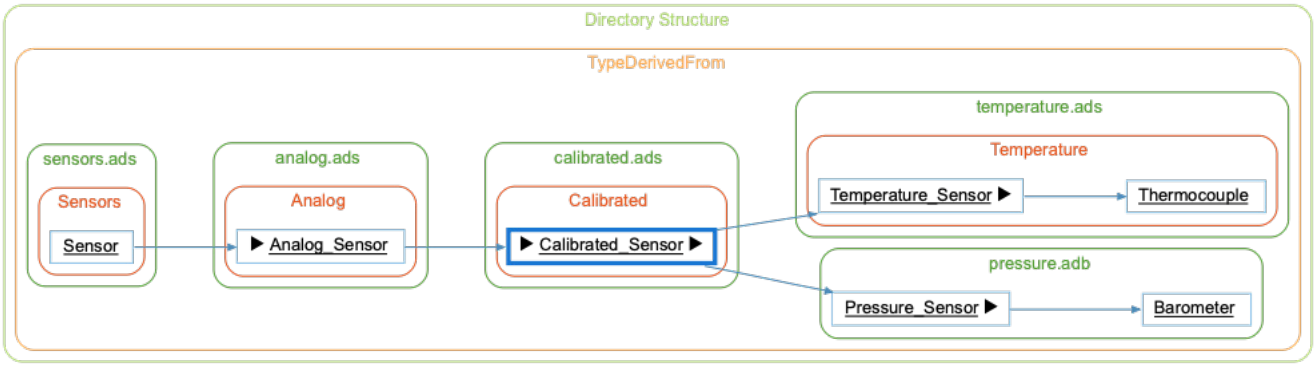
Back to Butterfly

Butterfly - Type Tree & Type Derived From (Cluster)

Show both Type Tree and Type Derived From relationships

See Type Tree and Type Derived From graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



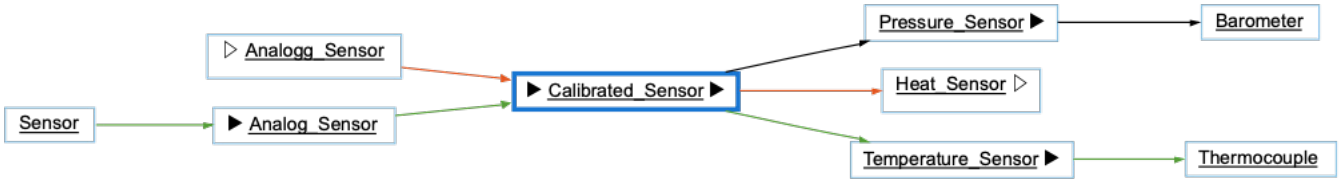
Back to Butterfly

Butterfly - Type Tree & Type Derived From (Compare)

Show both Type Tree and Type Derived From relationships

See Type Tree and Type Derived From graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Butterfly

Butterfly - Type Tree & Type Derived From (Simplified)

Show both Type Tree and Type Derived From relationships

See Type Tree and Type Derived From graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



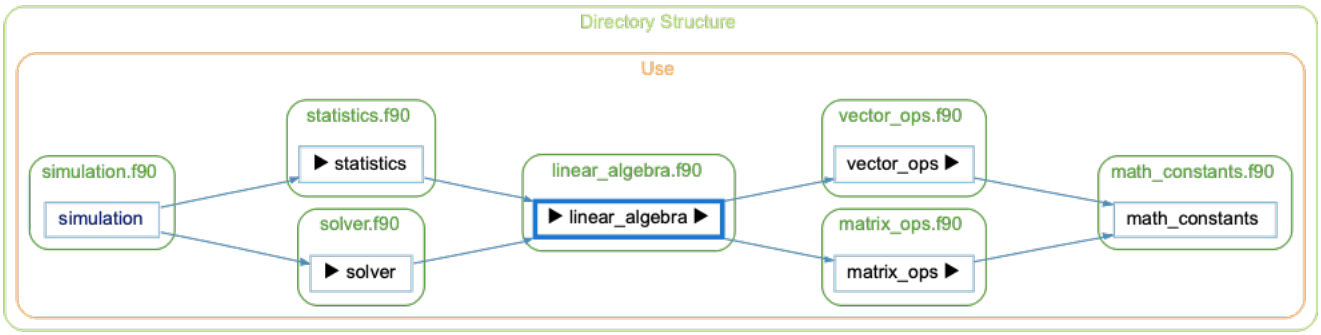
Back to Butterfly

Butterfly - Uses & Used By (Cluster)

Show both Uses and Used By relationships

See Uses and Used By graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



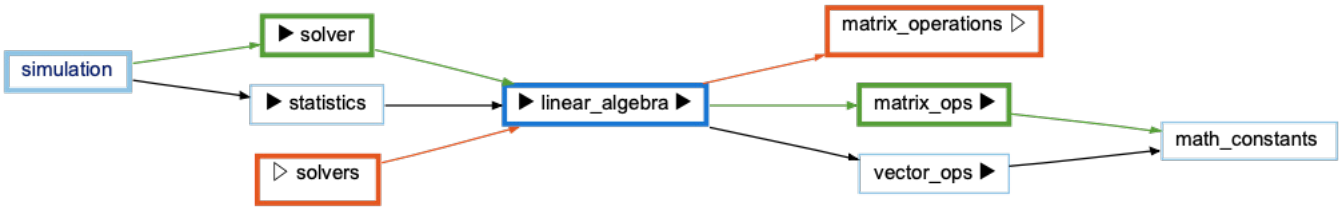
Back to Butterfly

Butterfly - Uses & Used By (Compare)

Show both Uses and Used By relationships

See Uses and Used By graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



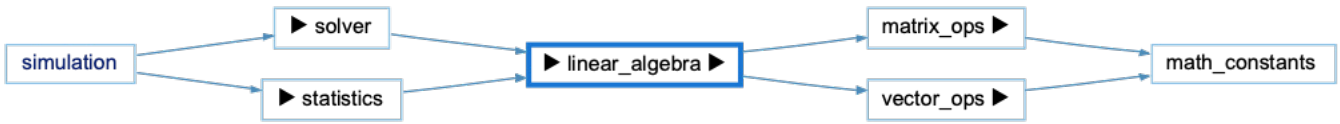
Back to Butterfly

Butterfly - Uses & Used By (Simplified)

Show both Uses and Used By relationships

See Uses and Used By graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



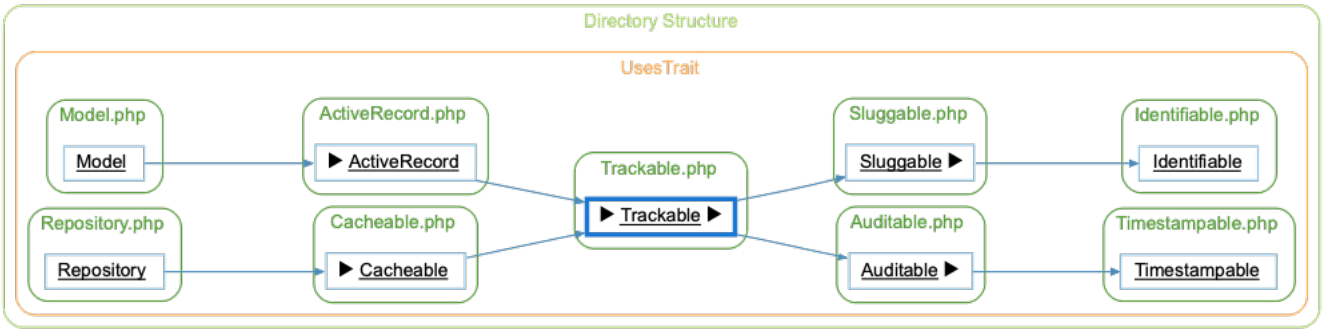
Back to Butterfly

Butterfly - Uses Traits & Used By (Cluster)

Show both Uses Traits and Used By relationships

See Uses Traits and Used By graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



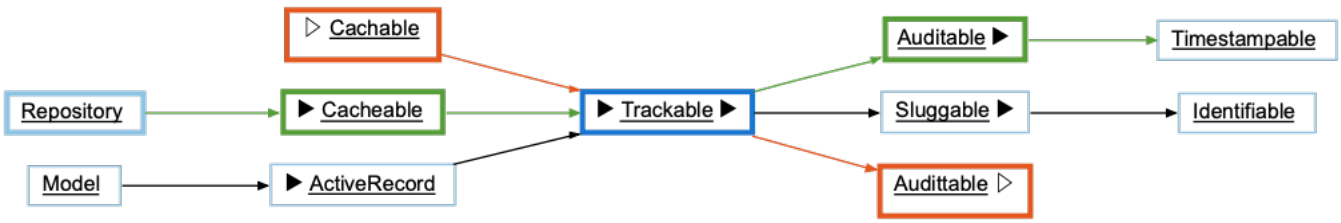
Back to Butterfly

Butterfly - Uses Traits & Used By (Compare)

Show both Uses Traits and Used By relationships

See Uses Traits and Used By graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



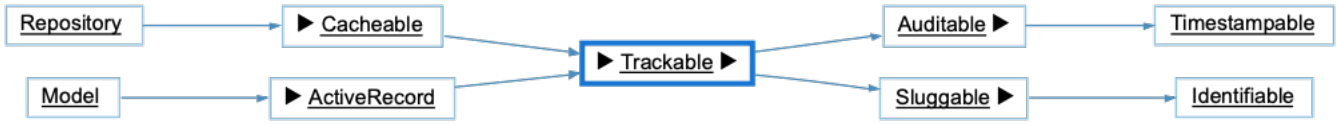
Back to Butterfly

Butterfly - Uses Traits & Used By (Simplified)

Show both Uses Traits and Used By relationships

See Uses Traits and Used By graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



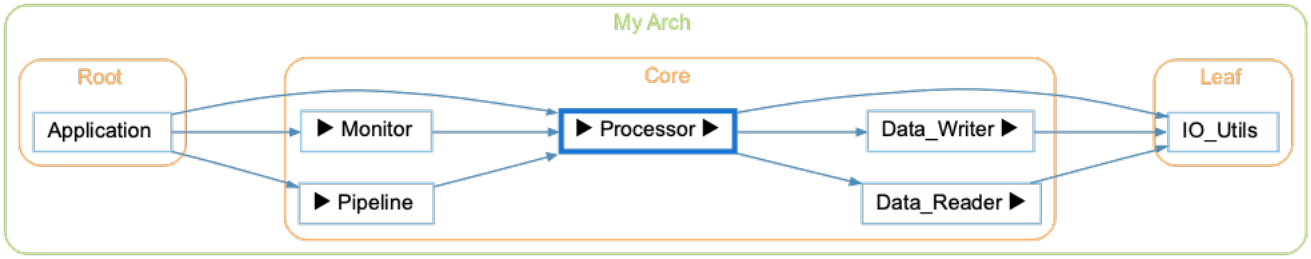
Back to Butterfly

Butterfly - With & Withby (Cluster)

Show both With and Withby relationships

See With and Withby graphs for details

The Cluster variant groups entities by their containing class/file/architecture.



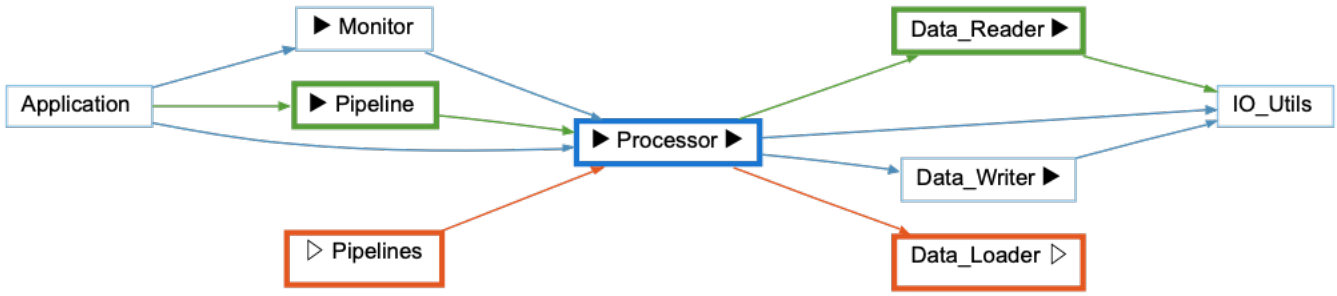
Back to Butterfly

Butterfly - With & Withby (Compare)

Show both With and Withby relationships

See With and Withby graphs for details

The Compare variant shows how the graph has changed relative to the project's comparison database.



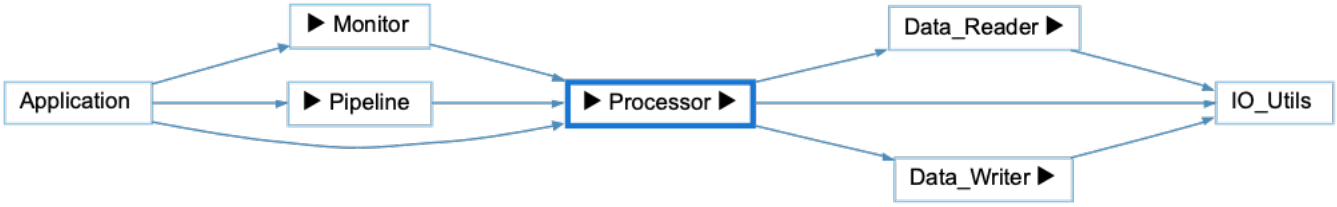
Back to Butterfly

Butterfly - With & Withby (Simplified)

Show both With and Withby relationships

See With and Withby graphs for details

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Butterfly

Called By

Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Functions, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show what calls a selected function, directly or indirectly, including through virtual dispatch.

For a selected function, this shows an edge from each function that calls it, then recurses into whatever calls those functions in turn, building a multi-level tree of callers.

Whenever a call could have come from more than one target at runtime — like through virtual dispatch or a function pointer — the graph shows all possible targets.

Rooted at callDemo below, the graph shows it's called directly by CallbyDerivedA::pureVirtualCall and by altFirst. CallbyDerivedA::pureVirtualCall is itself called directly by first, and is also reachable from first through a virtual call on CallbyBase::pureVirtualCall. The inverse graph, Calls, shows the reverse relationship — everything the selected function calls.

See the following code and corresponding graph

```
1 void callDemo(int n);
2
3 class CallbyBase {
4 public:
5   virtual void pureVirtualCall() = 0;
6 };
```

```
7
8 class CallbyDerivedA : public CallbyBase {
9 public:
10   void pureVirtualCall() override { callDemo(0); };
11 };
12
13 class CallbyDerivedB : public CallbyBase {
14 public:
15   void pureVirtualCall() override {};
16 };
17
18 void first()
19 {
20   CallbyDerivedA obj;
21   obj.pureVirtualCall(); // direct call
22
23   CallbyBase *base = &obj;
24   base->pureVirtualCall(); // virtual call
25 }
26
27 void altFirst()
28 {
29   callDemo(1);
30 }
```

Called By - Classic

The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Called By

Called By - Cluster

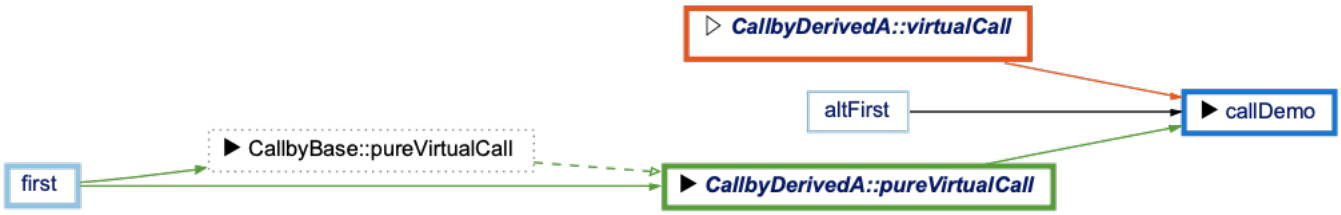
The Cluster variant groups entities by their containing class/file/architecture.



Back to Called By

Called By - Compare

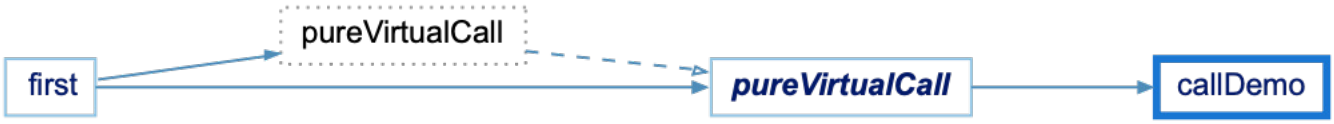
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Called By

Called By - Relationship

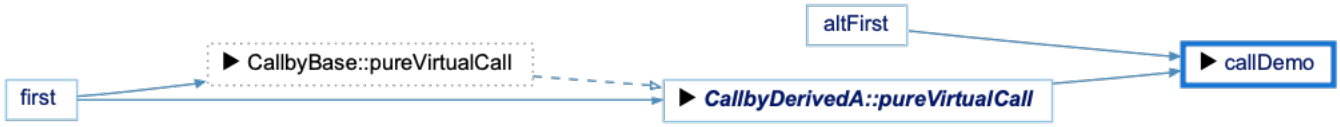
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Called By

Called By - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Called By

Calls

Languages: Ada, Any, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Functions, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified, With Global Objects (Classic)

Show what a selected function calls, directly or indirectly, including through function pointers, lambdas, and virtual dispatch.

For a selected function, this shows an edge to each function it calls, then recurses into each of those functions' own calls, building a multi-level call tree.

Whenever a call could have more than one target at runtime — like through virtual dispatch or a function pointer — the graph shows all possible targets. Some variants add extra edges showing how those targets were identified: an edge to a function used as a pointer (e.g. from functionPtrDemo to assignedToFunctionPtr), an edge from the variable to a function it's assigned to (e.g. from funcPtr to assignedToFunctionPtr), or an edge from a virtual function to each function that overrides it (e.g. from ComplexBase::virtualCall to ComplexDerive::virtualCall).

Rooted at callDemo below, the graph shows calls to directRecursive (which calls itself), indirectRecursiveA (which cycles with indirectRecursiveB), functionPtrDemo (which calls assignedToFunctionPtr through a function pointer), simpleOverridesDemo and complexOverridesDemo (each calling a virtual function through a base pointer), and lambdaDemo (which calls a lambda). The inverse graph, Called By, shows the reverse relationship — everything that calls the selected function.

See the following code and corresponding graph

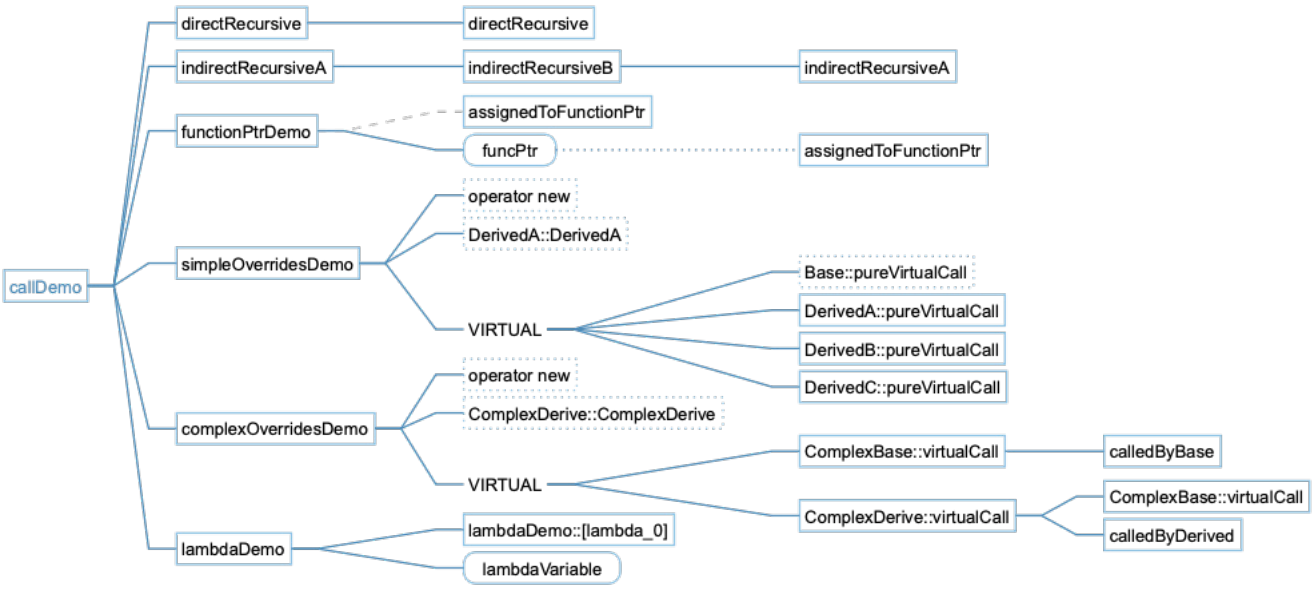
```
1 class Base {
2 public:
3   virtual void pureVirtualCall() = 0;
4 };
5
6 class DerivedA : public Base {
7 public:
8   void pureVirtualCall() override {}
9 };
10
11 class DerivedB : public Base {
12 public:
13   void pureVirtualCall() override {}
14 };
15
16 class DerivedC : public Base {
17 public:
18   void pureVirtualCall() override {}
19 };
20
21 class ComplexBase {
22 public:
23   virtual int virtualCall();
24 };
25
26 class ComplexDerive: public ComplexBase {
27 public:
28   int virtualCall() override;
29 };
```

```
30
31 int calledByBase() { return 0; }
32 int calledByDerived() { return 0; }
33
34 int ComplexBase::virtualCall() { return calledByBase(); }
35
36 int ComplexDerive::virtualCall()
37 {
38   auto val = ComplexBase::virtualCall();
39   return val + calledByDerived();
40 }
41
42 int GlobalObject = 0;
43
44 int directRecursive(int n) { return directRecursive(n); }
45 int indirectRecursiveA(int n);
46 int indirectRecursiveB(int n) { return indirectRecursiveA(n); }
47 int indirectRecursiveA(int n) { return indirectRecursiveB(n); }
48
49 int assignedToFunctionPtr(int x) { return x; }
50 int functionPtrDemo(int n)
51 {
52   if (GlobalObject > n)
53     return n;
54   int (*funcPtr)(int) = assignedToFunctionPtr;
55   return funcPtr(n); // function pointer call
56 }
57
58 void simpleOverridesDemo()
59 {
60   Base *base = new DerivedA;
61   base->pureVirtualCall();
62   GlobalObject ++;
63 }
64
65 void complexOverridesDemo()
66 {
67   ComplexBase *complexBase = new ComplexDerive;
68   complexBase->virtualCall();
69   GlobalObject = 0;
70 }
71
72 void lambdaDemo(int n)
73 {
74   auto lambdaVariable = [](int x) { return x * 3; };
75   lambdaVariable(n);
76 }
77
78 void callDemo(int n)
79 {
80   directRecursive(n);
81   indirectRecursiveA(n);
82   functionPtrDemo(n);
```

```
83    simpleOverridesDemo();
84    complexOverridesDemo();
85    lambdaDemo(n);
86  }
```

Calls - Classic

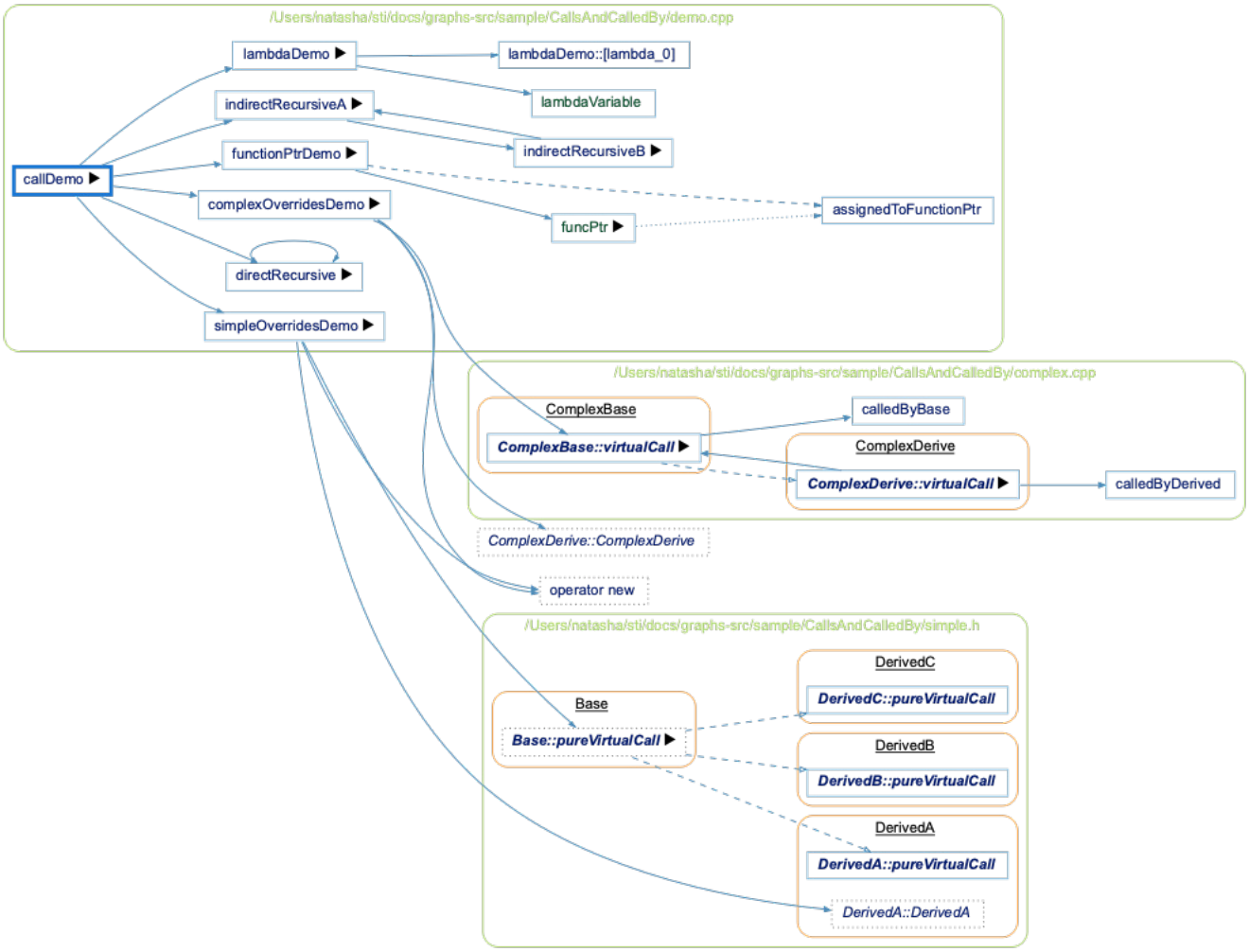
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Calls

Calls - Cluster

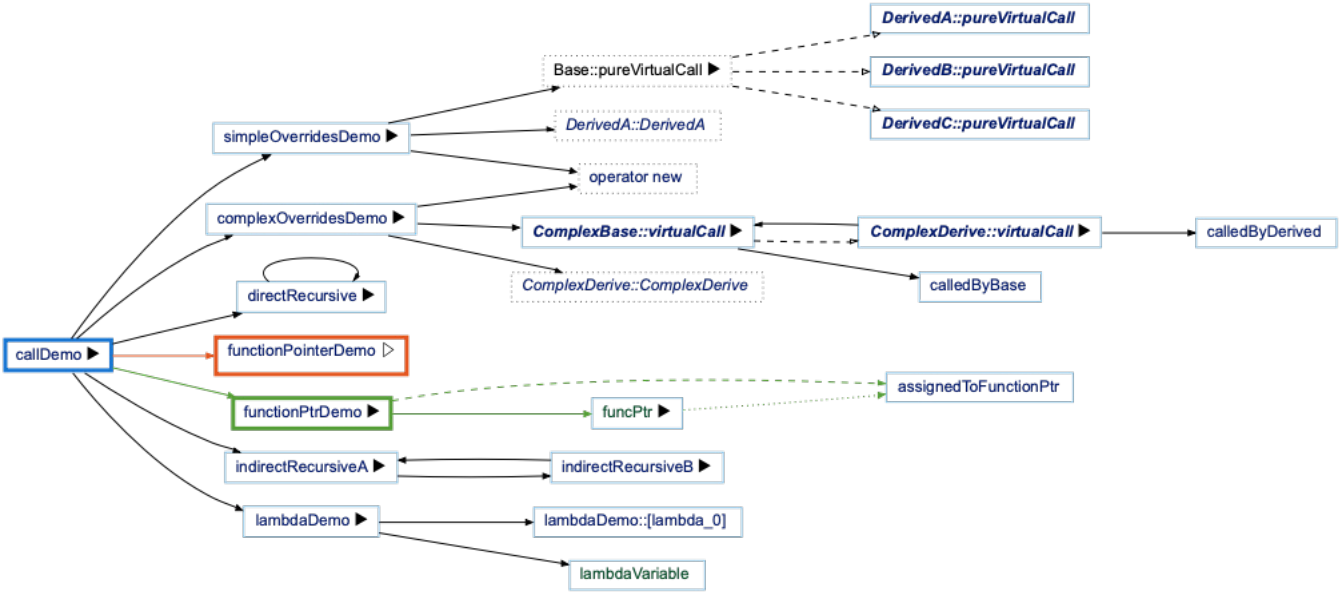
The Cluster variant groups entities by their containing class/file/architecture.



Back to Calls

Calls - Compare

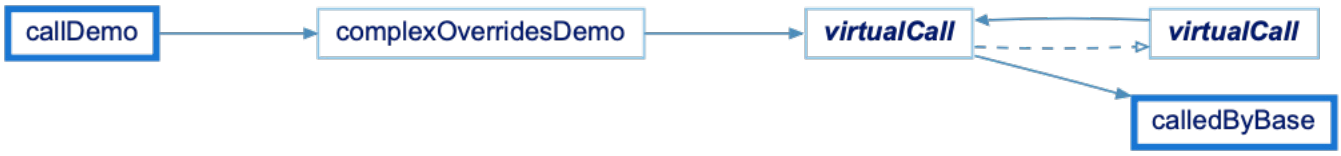
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Calls

Calls - Relationship

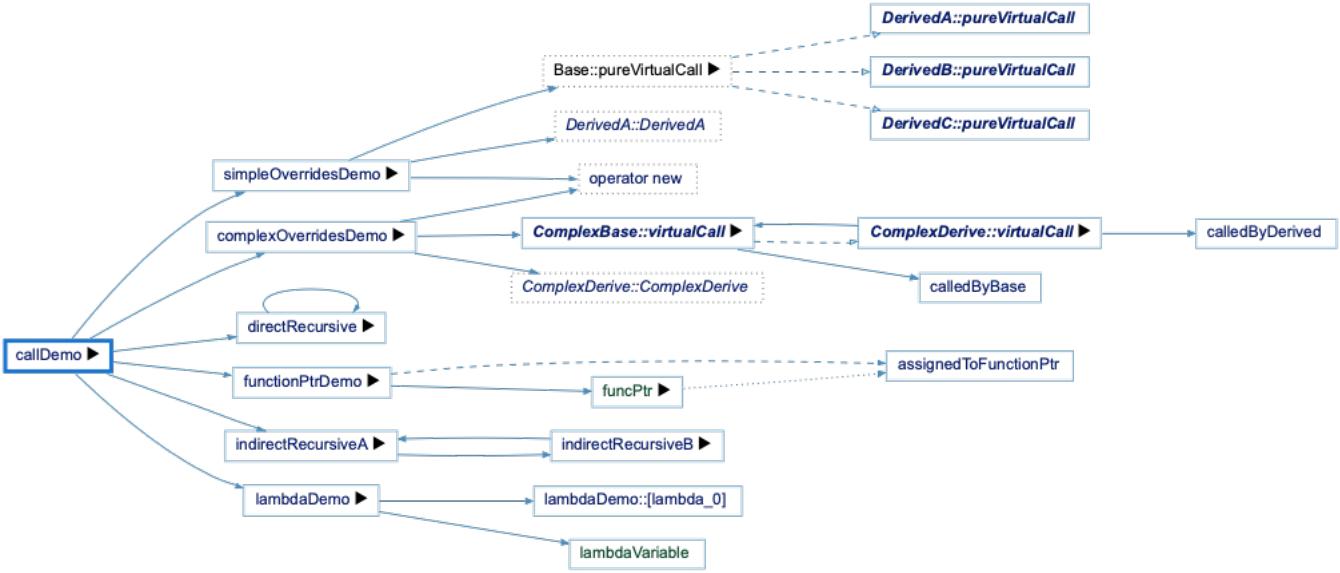
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Calls

Calls - Simplified

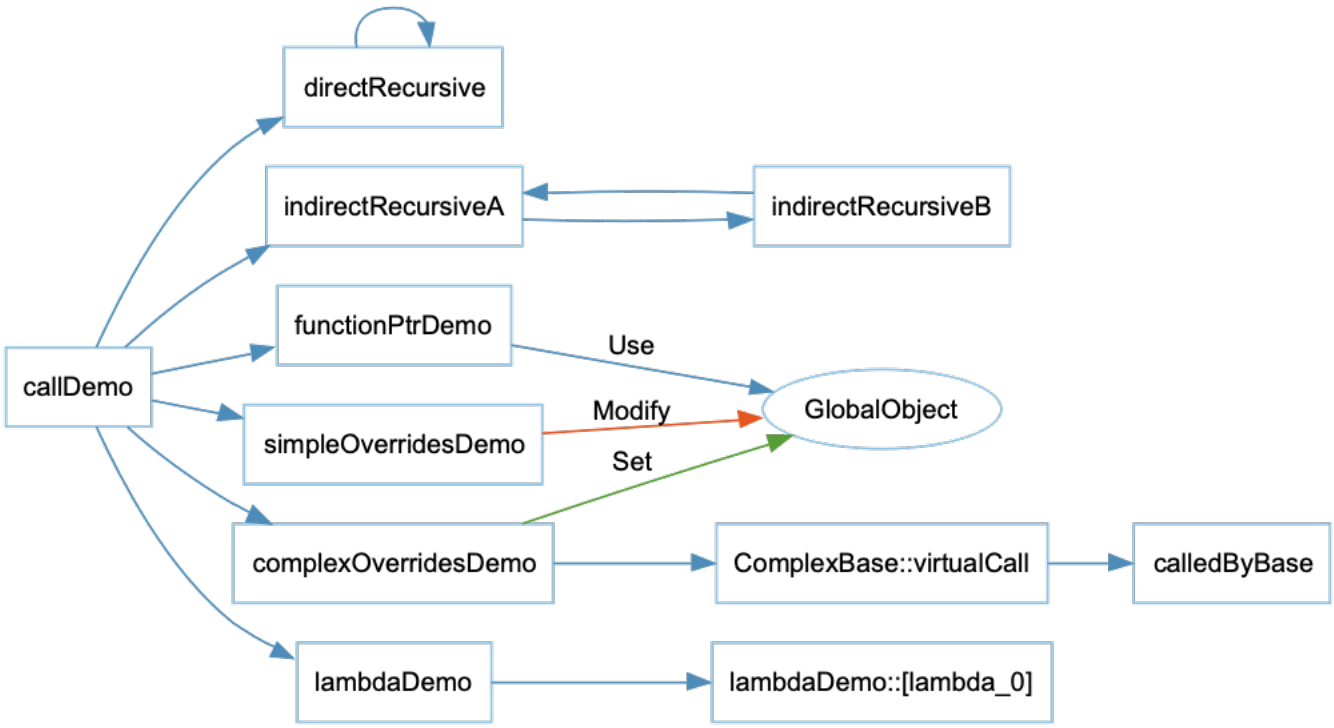
The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Calls

Calls - With Global Objects (Classic)

This variant also adds an edge from each function to any global object it sets, uses, or modifies, colored the same way as the Object References graph: a set is green, a modify is orange, and a use (or anything else) is the default blue. Below, functionPtrDemo uses GlobalObject, simpleOverridesDemo modifies it, and complexOverridesDemo sets it.



This variant is a plugin. It can be enabled/disabled from the Plugin Manager.

Back to Calls

Calls & Called By

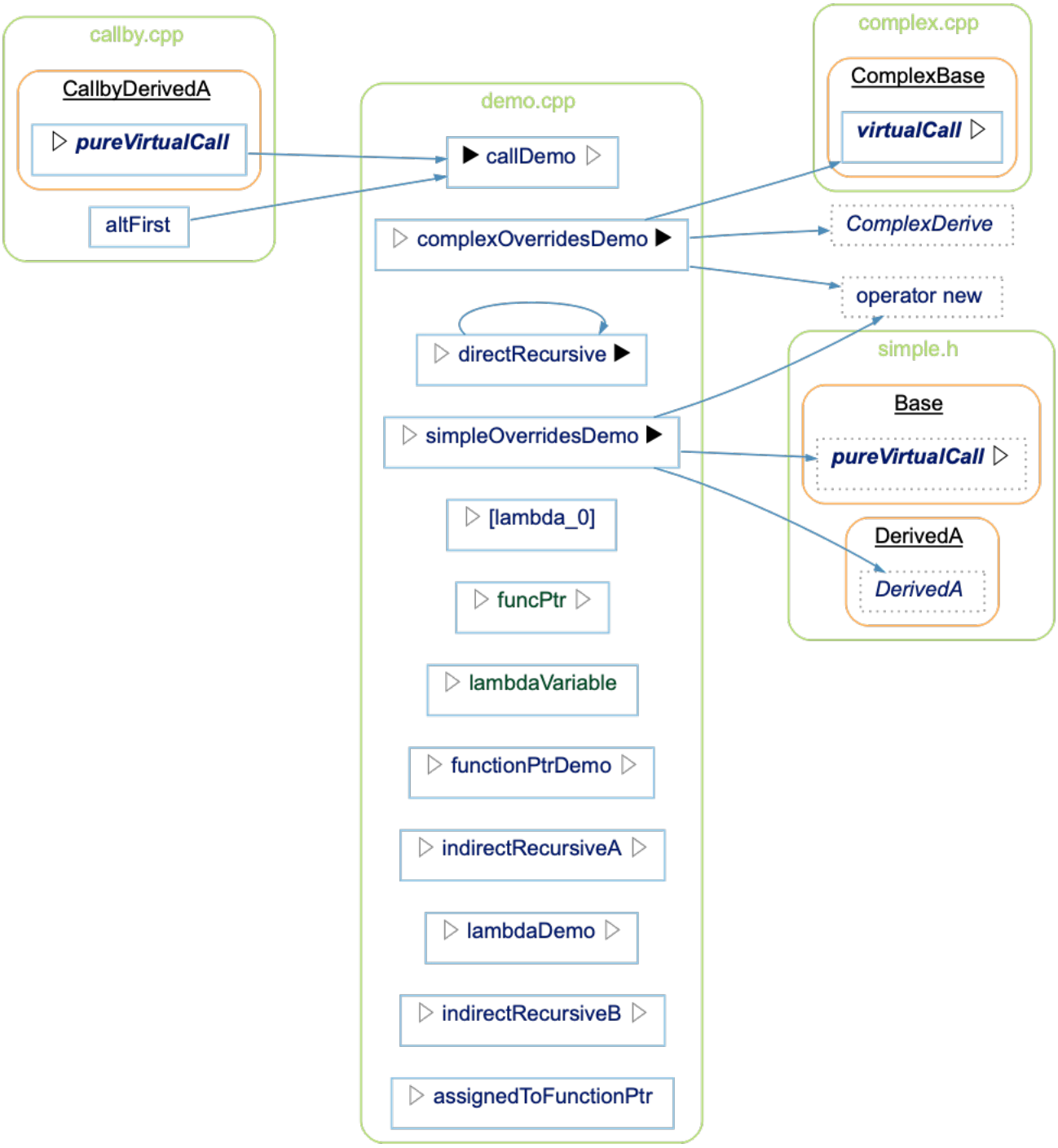
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Classes, Files
Variants: Butterfly, Called By, Calls, Internal

Show Calls and Called By relationships for a class, file, or architecture

See Calls and Called By graphs for details

Calls & Called By - Butterfly

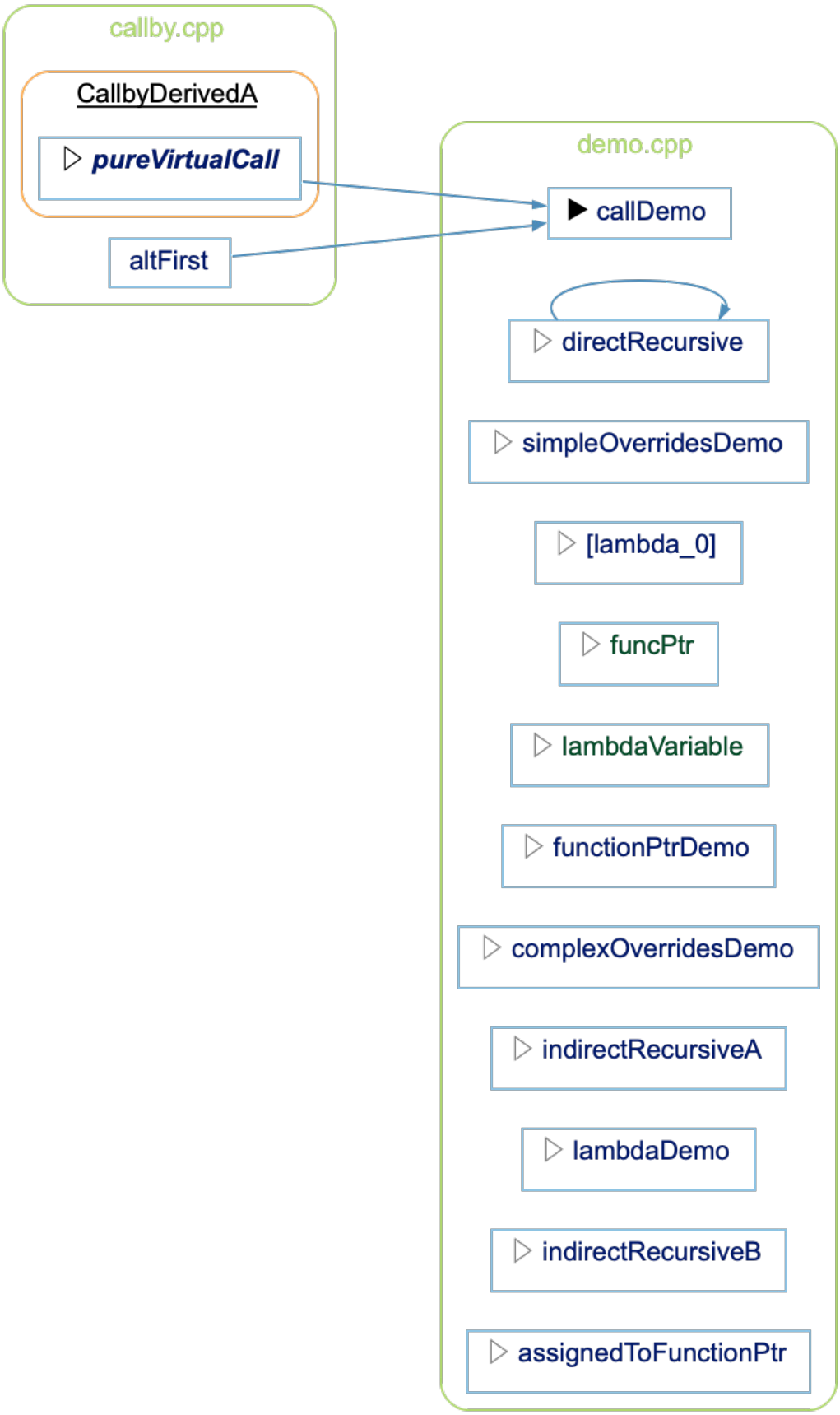
Show both Calls relationships from and Called By relationships to entities in a class, file, or architecture



Back to Calls & Called By

Calls & Called By - Called By

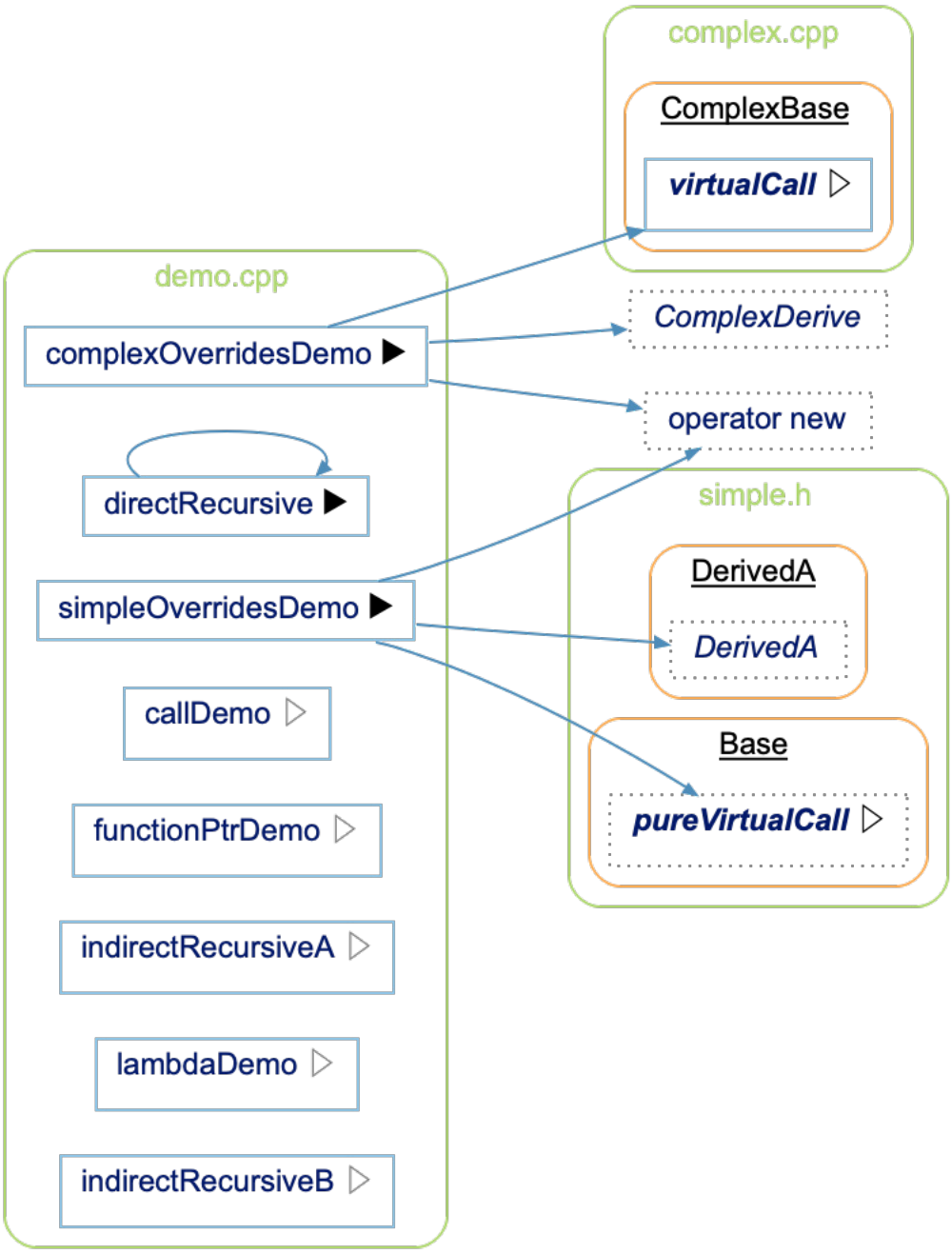
Show Called By relationships to entities in a class, file, or architecture


callby.cpp
CallbyDerivedA
pureVirtualCall
altFirst
demo.cpp
callDemo
directRecursive
simpleOverridesDemo
[lambda_0]
funcPtr
lambdaVariable
functionPtrDemo
complexOverridesDemo
indirectRecursiveA
lambdaDemo
indirectRecursiveB
assignedToFunctionPtr

Back to Calls & Called By

Calls & Called By - Calls

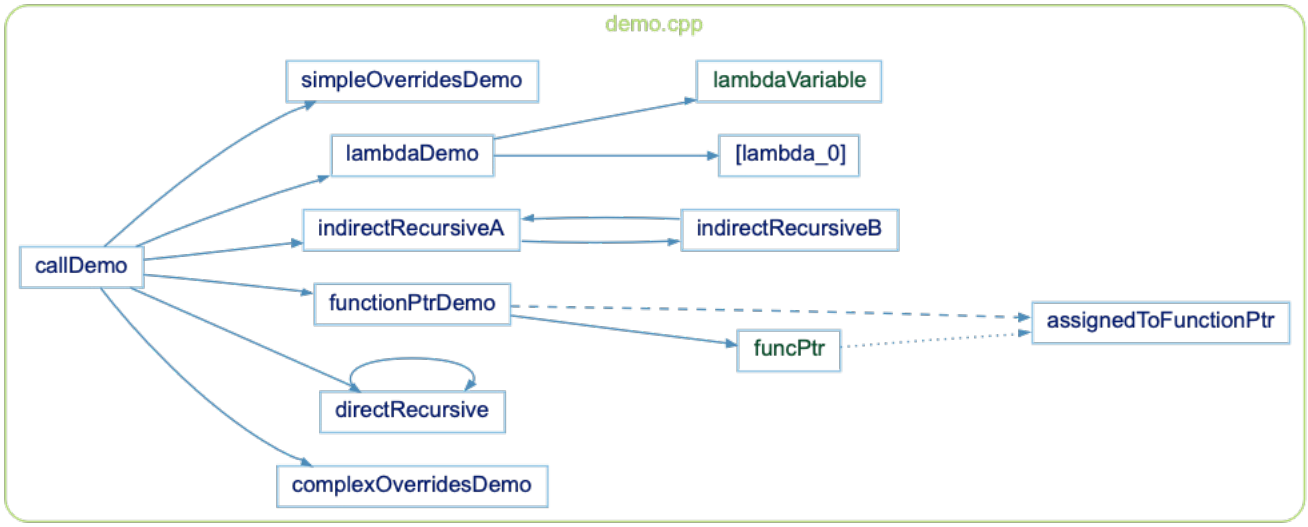
Show Calls relationships from entities in a class, file, or architecture



Back to Calls & Called By

Calls & Called By - Internal

Show Calls relationships between entities in a class, file, or architecture



Back to Calls & Called By

Child Lib Units

Languages: Ada
Targets: Packages
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the Ada library units declared directly or indirectly as children of a selected package.

For a selected package, this shows an edge to each library unit — a package, function, or procedure — declared under it using dotted (child unit) notation, then recurses into each child package's own children, building a multi-level tree. Only a package can have children, so a child function or procedure is always a leaf.

Rooted at Telemetry.Sensors.Analog below, the graph shows its four direct children: the packages Telemetry.Sensors.Analog.Temperature and Telemetry.Sensors.Analog.Pressure, and the subprograms Telemetry.Sensors.Analog.Sample and Telemetry.Sensors.Analog.Configure. The inverse graph, Parent Lib Unit, shows the reverse relationship — the package a selected library unit is declared under.

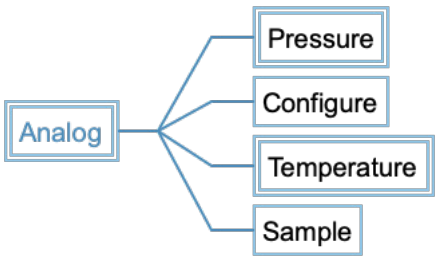
See the following code and corresponding graph

```
1 package Telemetry.Sensors.Analog is
2    Max_Channels : constant := 8;
3 end Telemetry.Sensors.Analog;
4
5 function Telemetry.Sensors.Analog.Sample
```

```
6     (Channel : Natural) return Float;
7
8  procedure Telemetry.Sensors.Analog.Configure
9     (Channel : Natural; Rate : Natural);
10
11 package Telemetry.Sensors.Analog.Temperature is
12    Offset : constant Float := 0.5;
13 end Telemetry.Sensors.Analog.Temperature;
14
15 package Telemetry.Sensors.Analog.Pressure is
16    Scale : constant Float := 100.0;
17 end Telemetry.Sensors.Analog.Pressure;
```

Child Lib Units - Classic

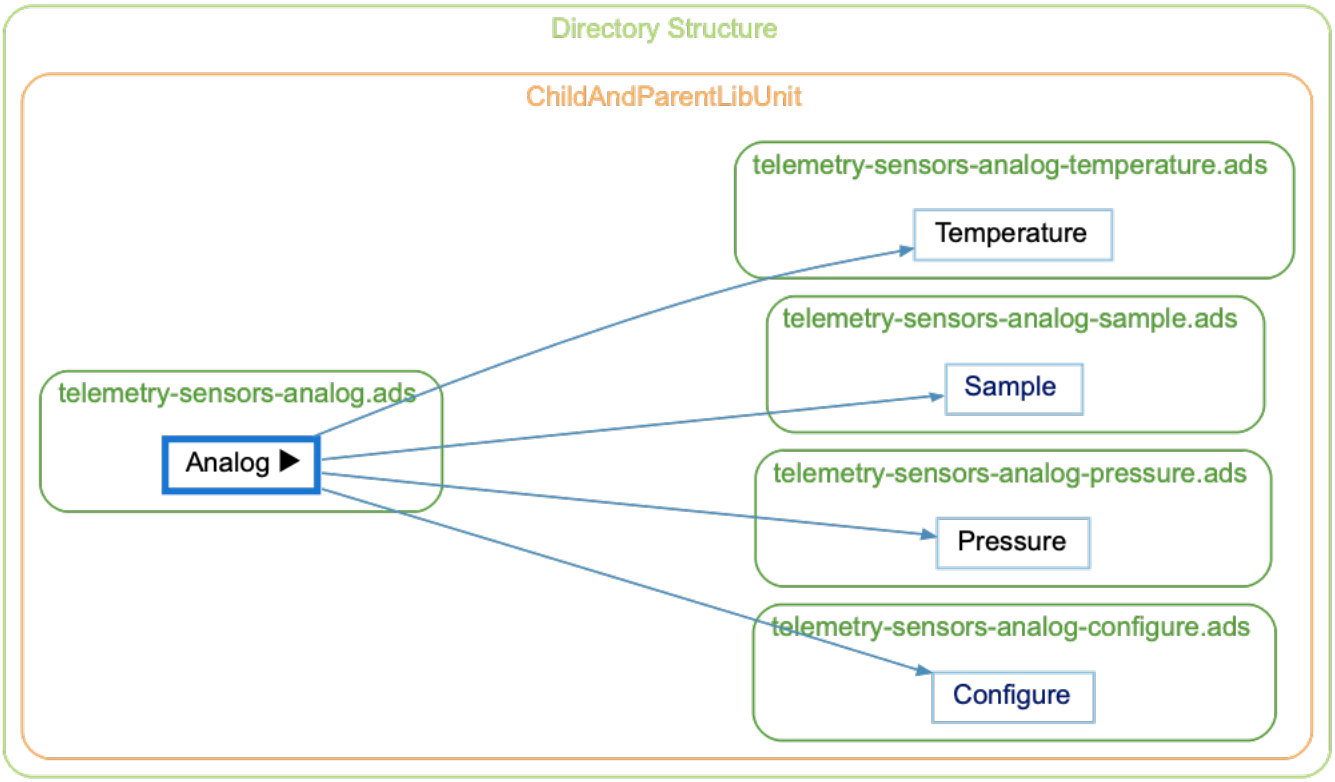
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Child Lib Units

Child Lib Units - Cluster

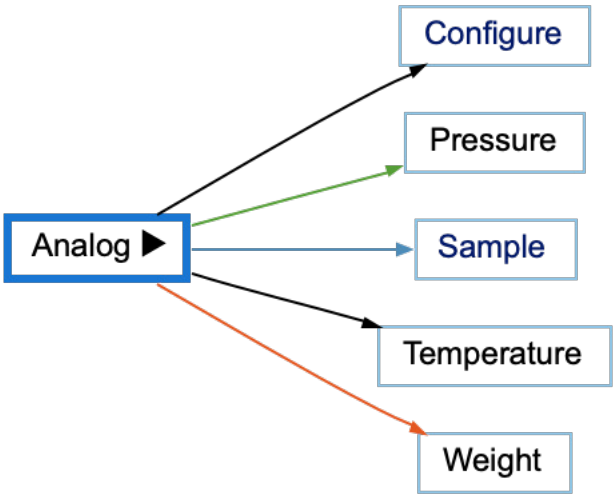
The Cluster variant groups entities by their containing class/file/architecture.



Back to Child Lib Units

Child Lib Units - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Child Lib Units

Child Lib Units - Relationship

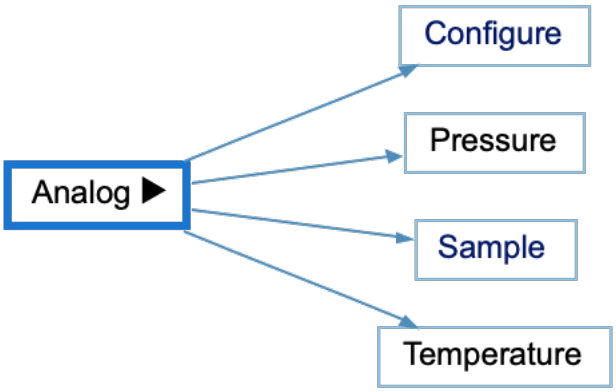
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Child Lib Units

Child Lib Units - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Child Lib Units

Child Lib Units & Parent Lib Unit

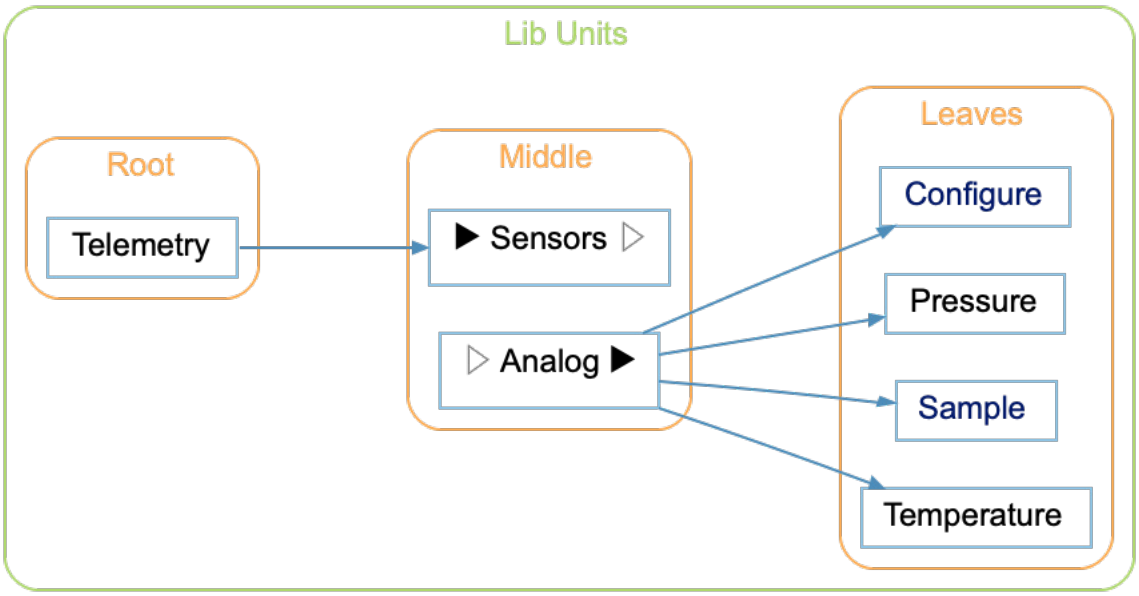
Languages: Ada
Targets: Architectures
Variants: Butterfly, Child Lib Units, Internal, Parent Lib Unit

Show Child Lib Units and Parent Lib Unit relationships for an architecture

See Child Lib Units and Parent Lib Unit graphs for details

Child Lib Units & Parent Lib Unit - Butterfly

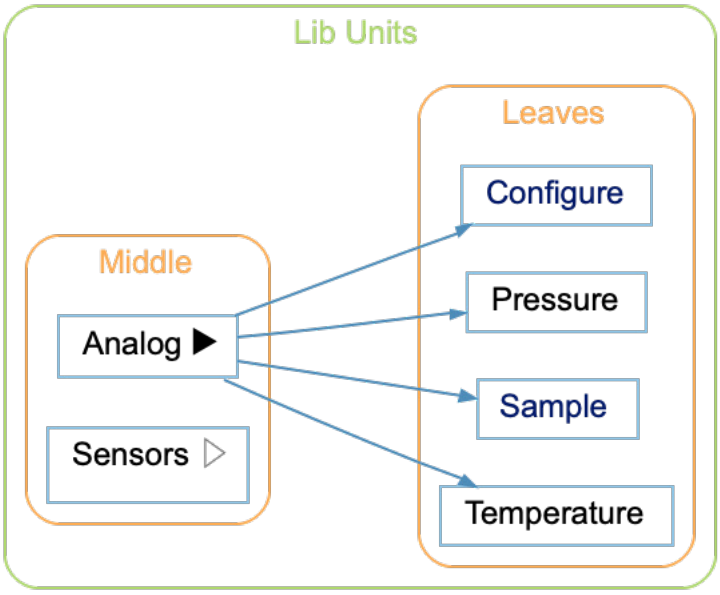
Show both Child Lib Units relationships from and Parent Lib Unit relationships to entities in an architecture



Back to Child Lib Units & Parent Lib Unit

Child Lib Units & Parent Lib Unit - Child Lib Units

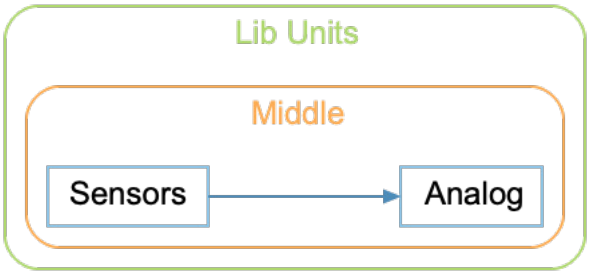
Show Child Lib Units relationships from entities in an architecture



Back to Child Lib Units & Parent Lib Unit

Child Lib Units & Parent Lib Unit - Internal

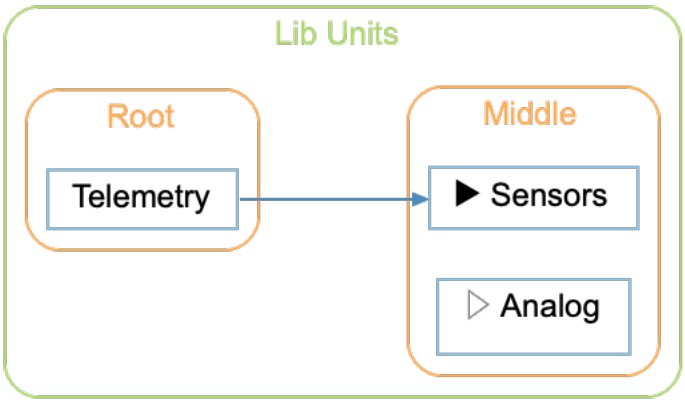
Show Child Lib Units relationships between entities in an architecture



Back to Child Lib Units & Parent Lib Unit

Child Lib Units & Parent Lib Unit - Parent Lib Unit

Show Parent Lib Unit relationships to entities in an architecture



Back to Child Lib Units & Parent Lib Unit

Contains

Languages: Java
Targets: Packages
Variants: Classic

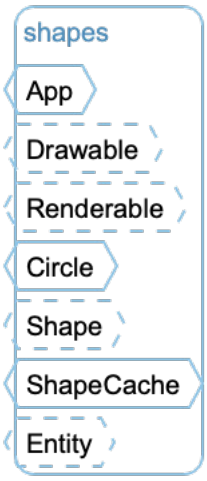
Show everything a selected Java package directly contains.

The main box lists every top-level type declared across the package's files, regardless of visibility.

Rooted at the package shapes below, the box shows the classes App, Shape (abstract), ShapeCache, Circle, and Entity (abstract), and the interfaces Drawable and Renderable.

See the following code and corresponding graph

```
1 // App.java
2 package shapes;
3
4 public class App {
5     /* ... */
6 }
7
8 // Shape.java
9 package shapes;
10
11 public abstract class Shape
12     extends Entity implements Drawable, Renderable {
13     /* ... */
14 }
15
16 class ShapeCache {
17     /* ... */
18 }
19
20 // Circle.java
21 package shapes;
22
23 public class Circle extends Shape {
24     /* ... */
25 }
26
27 // Entity.java
28 package shapes;
29
30 public abstract class Entity {
31     /* ... */
32 }
33
34 // Drawable.java
35 package shapes;
36
37 public interface Drawable {
38     /* ... */
39 }
40
41 interface Renderable {
42     /* ... */
43 }
```



Control Flow

Languages: Any
Targets: Functions, Subprograms
Variants: Classic, Compare

Show the flow of control through a selected function, method, or subprogram as a flowchart.

The default shapes and colors are:

- Conditionals (if/else if, and similar): diamond
- Loops (for, while, do while, and similar): oval
- catch/finally: oval
- try: hexagon
- switch/case (and similar): octagon
- Everything else: rectangle
- Unreachable code: red, regardless of shape

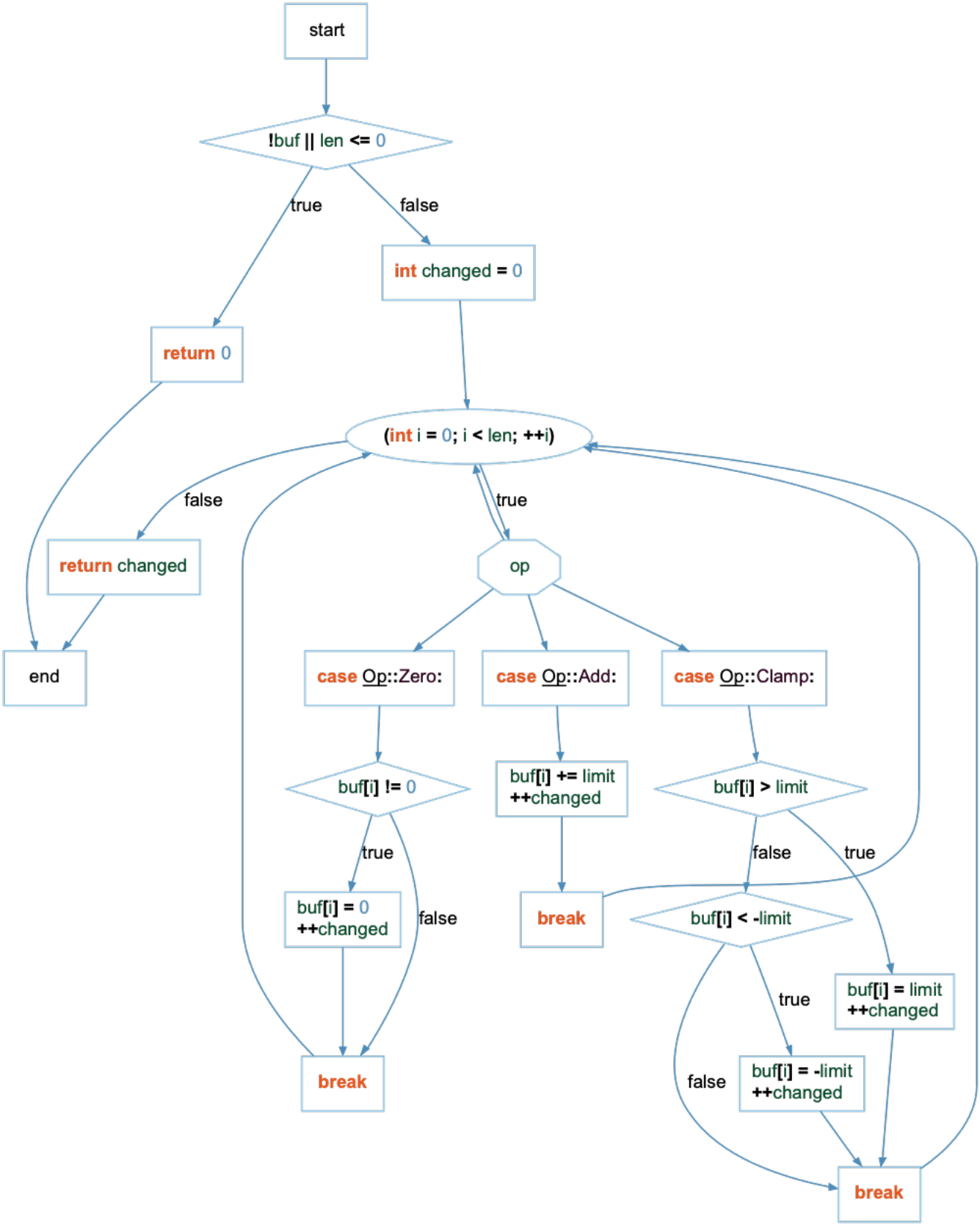
Edges leaving a conditional or loop are labeled for the branch taken.

For example, see the following code and corresponding graph

```
1 enum class Op { Add, Clamp, Zero };
2
3 int process(int* buf, int len, Op op, int limit) {
4     if (!buf || len <= 0)  // diamond: true returns early
5         return 0;
6
7     int changed = 0;
8     for (int i = 0; i < len; ++i) {  // oval: loop clause
9         switch (op) {  // octagon: edge to each case
10            case Op::Add:
```

```
11                  buf[i] += limit;
12                  ++changed;
13                  break;  // loops back to the oval above
14              case Op::Clamp:
15                  if (buf[i] > limit) {  // diamond
16                      buf[i] = limit;
17                      ++changed;
18                  } else if (buf[i] < -limit) {  // diamond
19                      buf[i] = -limit;
20                      ++changed;
21                  }
22                  break;  // loops back to the oval above
23              case Op::Zero:
24                  if (buf[i] != 0) {  // diamond
25                      buf[i] = 0;
26                      ++changed;
27                  }
28                  break;  // loops back to the oval above
29          }
30      }
31      return changed;
32  }
```

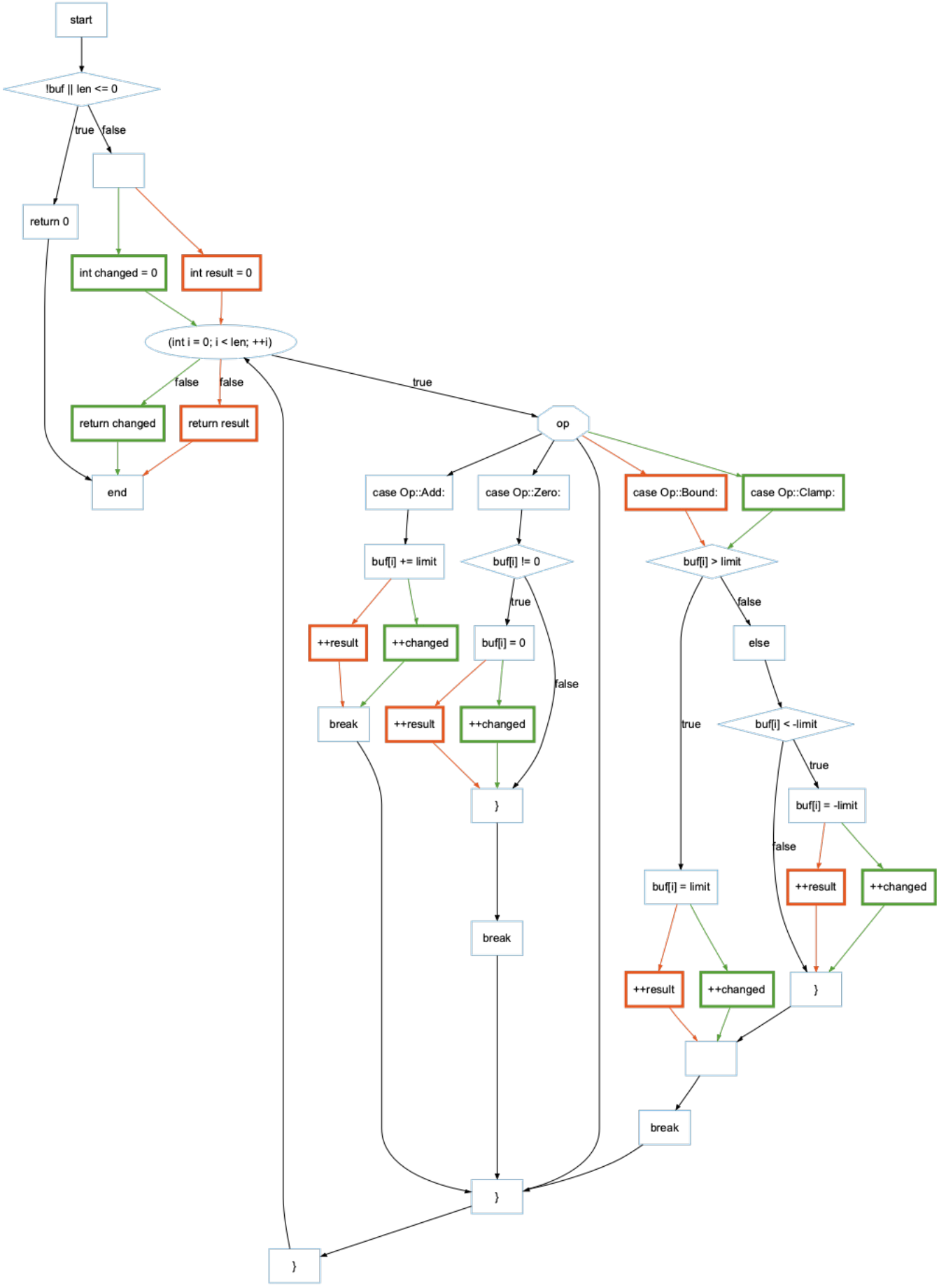
Control Flow - Classic



Back to Control Flow

Control Flow - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.


start
!buf || len <= 0
true
false
return 0
int changed = 0
int result = 0
(int i = 0; i < len; ++i)
false
false
true
return changed
return result
end
op
case Op::Add:
case Op::Zero:
case Op::Bound:
case Op::Clamp:
buf[i] += limit
buf[i] != 0
buf[i] > limit
true
false
++result
++changed
buf[i] = 0
else
false
break
++result
++changed
true
buf[i] < -limit
}
true
buf[i] = -limit
false
buf[i] = limit
++result
++changed
++result
++changed
}
break
break
}
}

Back to Control Flow

Data Components

Languages: Ada
Targets: Record Types
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the record and array types nested inside a selected record type, down to their primitive fields.

For a selected record type, this shows an edge to each of its components (fields), then an edge from that component to its own type — another record type or an array type — and recurses into that type's components in turn. An array component's node shows its bounds and element type, and if that element type is itself a record or array, the graph continues into its components too. A component whose type has no further components of its own, like a Float, is a leaf.

Rooted at Route below, the graph shows its components Path (a Navigation.Segment) and Stops (a Navigation.Waypoint_Array). Segment continues into its own components From, To, and Speeds; Waypoint_Array continues into its element type, Coordinate, which is shown again with its own X and Y components.

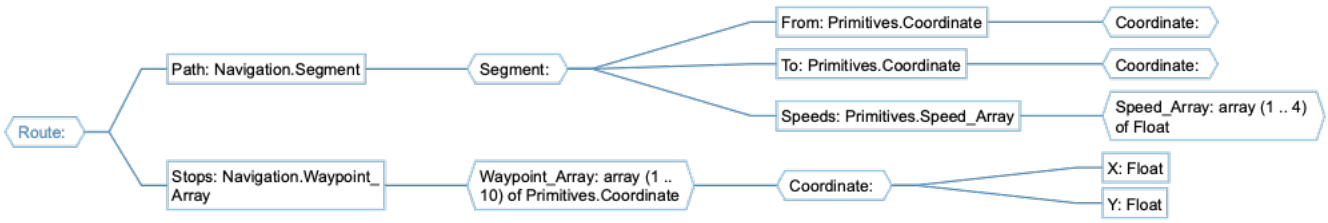
See the following code and corresponding graph

```
1 package Primitives is
2    type Coordinate is record
3       X, Y : Float;
4    end record;
5
6    type Speed_Array is array (1 .. 4) of Float;
7 end Primitives;
8
9 package Navigation is
10    type Segment is record
11       From   : Primitives.Coordinate;
12       To     : Primitives.Coordinate;
13       Speeds : Primitives.Speed_Array;
14    end record;
15
16    type Waypoint_Array is array (1 .. 10) of Primitives.Coordinate;
17 end Navigation;
18
19 package Routes is
20    type Route is record
21       Path  : Navigation.Segment;
22       Stops : Navigation.Waypoint_Array;
23    end record;
```

```
24 end Routes;
```

Data Components - Classic

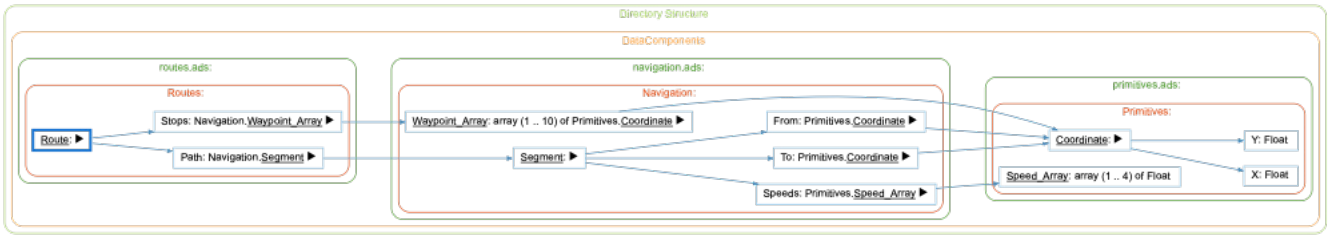
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Data Components

Data Components - Cluster

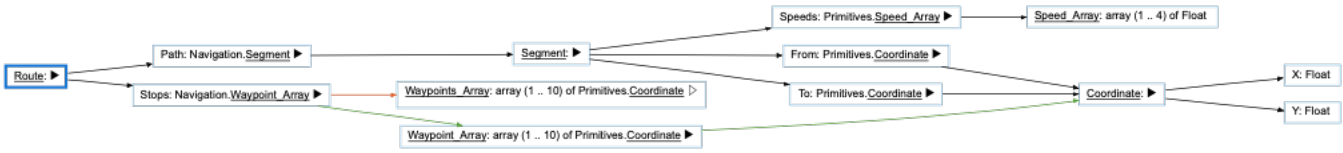
The Cluster variant groups entities by their containing class/file/architecture.



Back to Data Components

Data Components - Compare

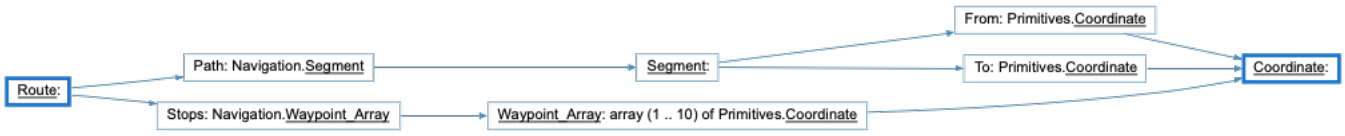
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Data Components

Data Components - Relationship

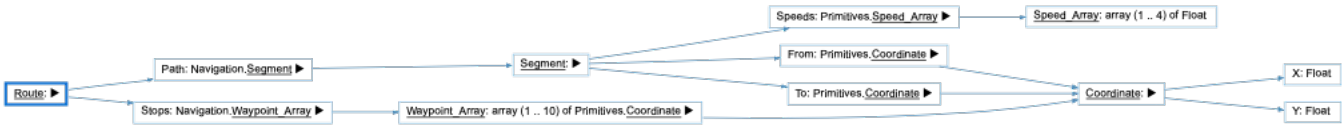
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Data Components

Data Components - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Data Components

Data Flow In

Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Objects
Variants: Cluster, Compare, Relationship, Simplified

Show what sets or modifies a selected object's value, directly or indirectly.

For a selected object, variable, field, or property, this shows an edge to each function that sets or modifies it, then recurses into whatever objects that function uses, building a multi-level tree back through the computation. Each function in the tree also shows its own callers.

Rooted at transformedData below, the graph shows it's set by both transform and normalize. transform uses filteredData and auxData, which are set by filter and loadAux in turn; filter uses rawInput, set by acquire. normalize also uses auxData, and reads transformedData's own current value before conditionally overwriting it. pipeline, which calls acquire, loadAux, filter, and transform, appears throughout the tree as each one's caller. The inverse graph, Data Flow Out, shows the reverse relationship — everywhere a selected object's value is used.

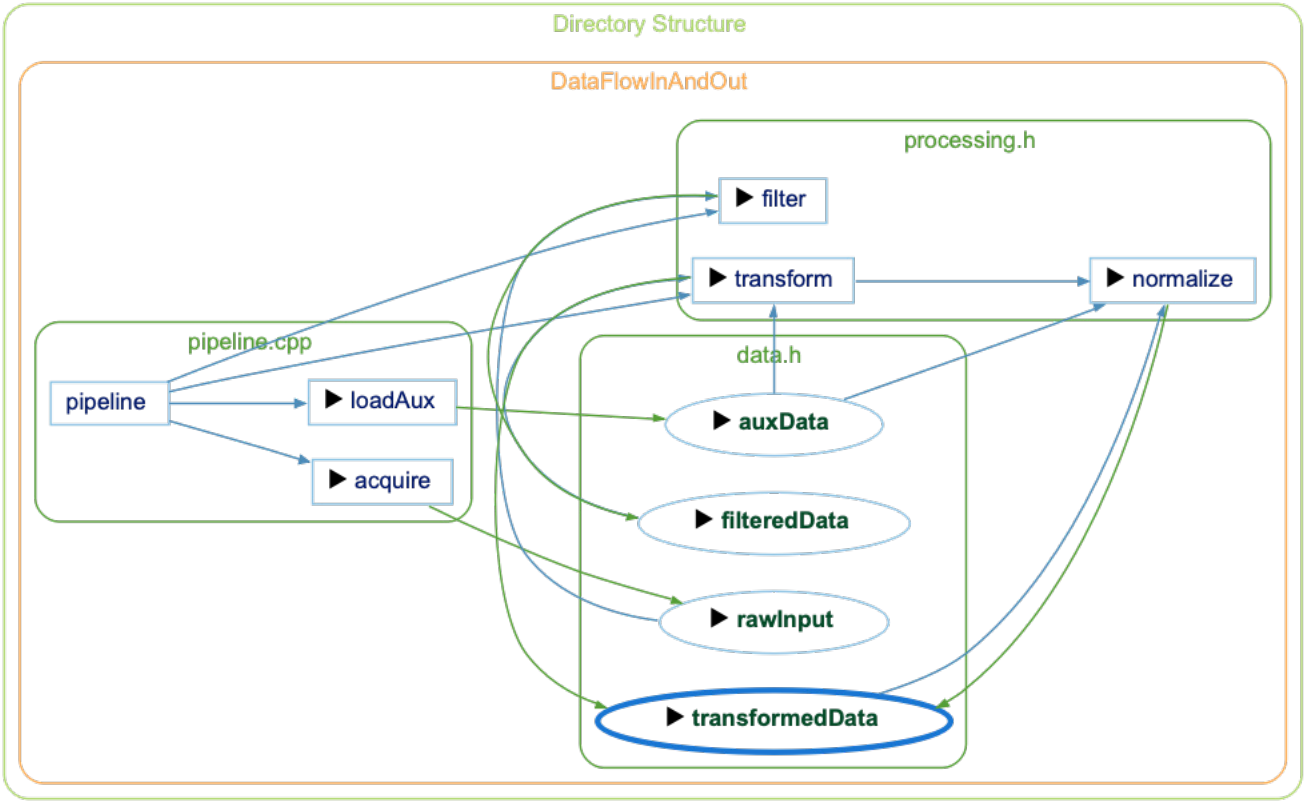
See the following code and corresponding graph

```
1 extern int rawInput;
2 extern int auxData;
3 extern int filteredData;
4 extern int transformedData;
5 extern int logBuffer;
6
```

```
7 void output();
8 void logData();
9 void accumulate();
10 void report();
11
12 void acquire()
13 {
14     rawInput = 42;
15 }
16
17 void loadAux()
18 {
19     auxData = 10;
20 }
21
22 void filter(int threshold)
23 {
24     filteredData = (rawInput > threshold) ? rawInput : 0;
25 }
26
27 void transform()
28 {
29     transformedData = filteredData * auxData;
30     logBuffer       = filteredData;
31     normalize();
32 }
33
34 void normalize()
35 {
36     if (transformedData > 100)
37         transformedData = 100;
38     else
39         transformedData = transformedData + auxData;
40 }
41
42 void pipeline(int threshold)
43 {
44     acquire();
45     loadAux();
46     filter(threshold);
47     transform();
48     output();
49     logData();
50     accumulate();
51     report();
52 }
```

Data Flow In - Cluster

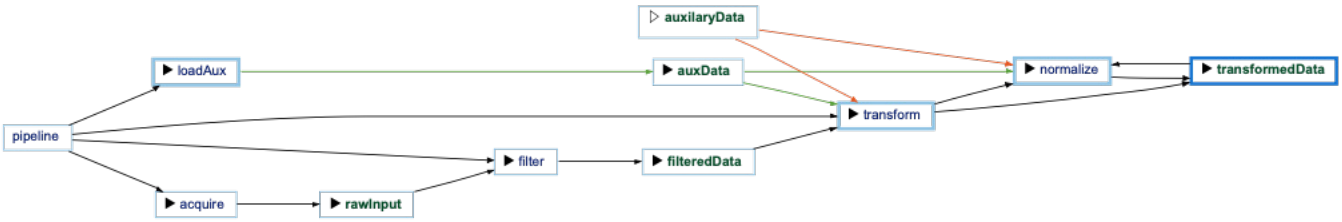
The Cluster variant groups entities by their containing class/file/architecture.



Back to Data Flow In

Data Flow In - Compare

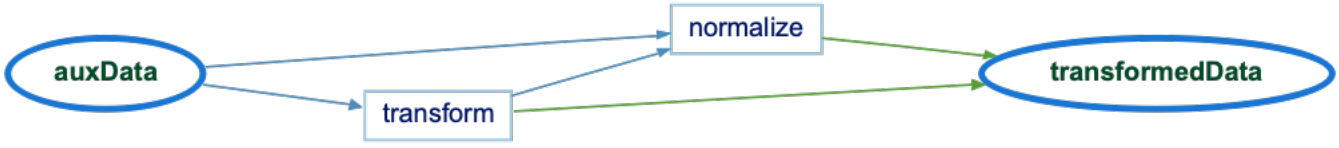
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Data Flow In

Data Flow In - Relationship

The Relationship variant filters the graph to only paths connecting the two entities.



Back to Data Flow In

Data Flow In - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Data Flow In

Data Flow Out

Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Objects
Variants: Cluster, Compare, Relationship, Simplified

Show what uses a selected object's value, directly or indirectly.

For a selected object, variable, field, or property, this shows an edge to each function that uses it, then recurses into whatever objects that function sets or modifies, continuing to whoever uses those objects in turn, building a multi-level tree forward through the computation. Each function in the tree also shows its own callees.

Rooted at transformedData below, the graph shows it's used by normalize (which also sets it, reading its own current value first), output, and logData. output sets outputResult, which is used by accumulate, which modifies runningTotal, which is used by report which sets reportValue. logData modifies logBuffer rather than replacing it outright. The inverse graph, Data Flow In, shows the reverse relationship — everything that sets or modifies a selected object's value.

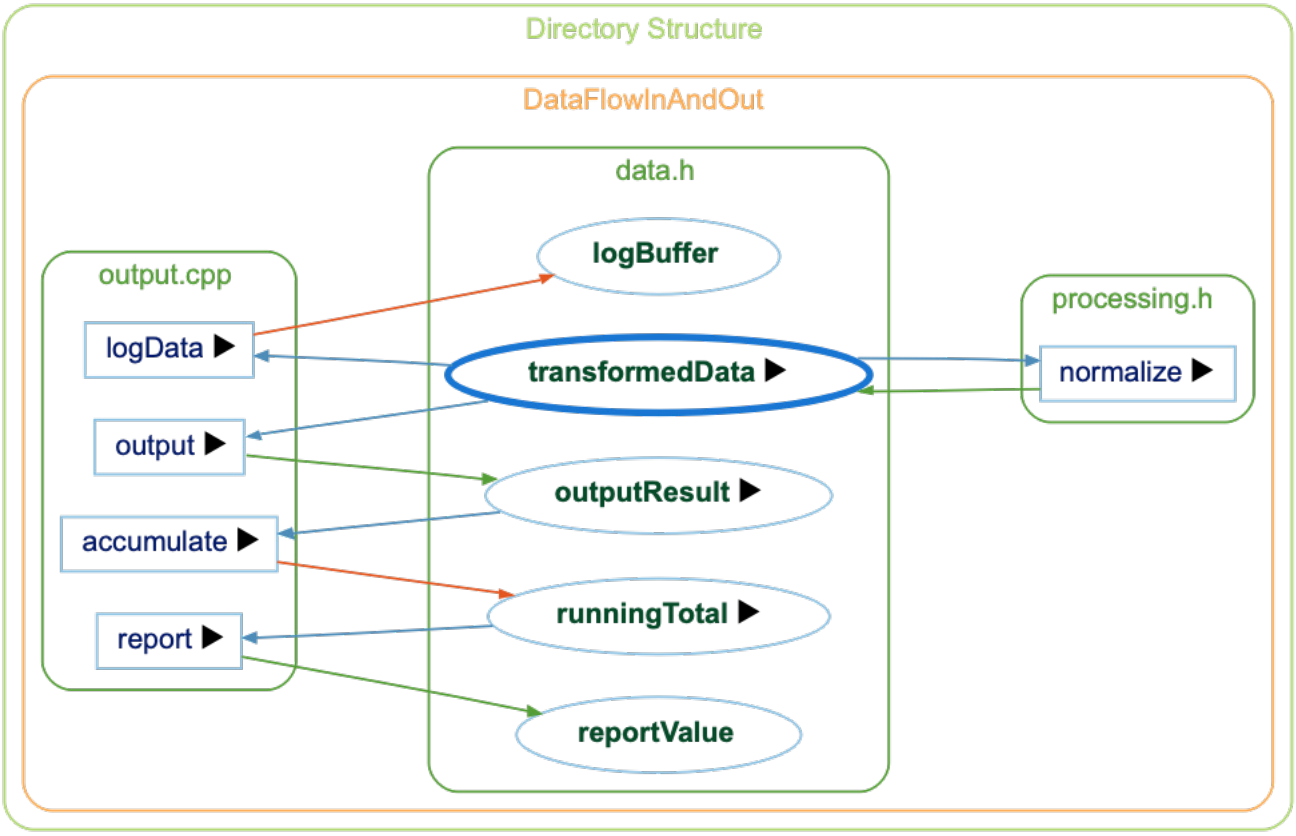
See the following code and corresponding graph

```
1 extern int auxData;
2 extern int transformedData;
3 extern int outputResult;
4 extern int logBuffer;
5 extern int runningTotal;
6 extern int reportValue;
7
8 void normalize()
9 {
```

```
10      if (transformedData > 100)
11          transformedData = 100;
12      else
13          transformedData = transformedData + auxData;
14  }
15
16  void output()
17  {
18      outputResult = transformedData;
19  }
20
21  void logData()
22  {
23      logBuffer += transformedData;
24  }
25
26  void accumulate()
27  {
28      runningTotal += outputResult;
29  }
30
31  void report()
32  {
33      reportValue = runningTotal;
34  }
```

Data Flow Out - Cluster

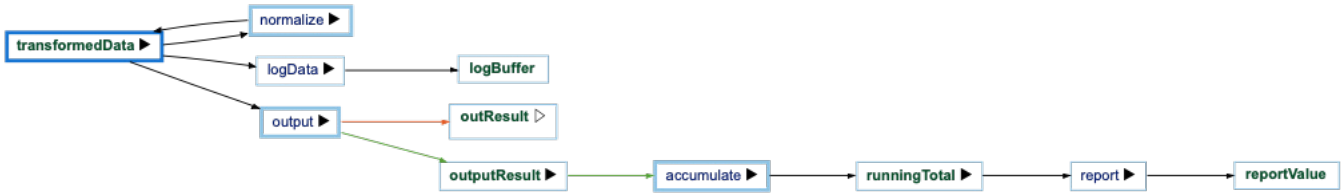
The Cluster variant groups entities by their containing class/file/architecture.



Back to Data Flow Out

Data Flow Out - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Data Flow Out

Data Flow Out - Relationship

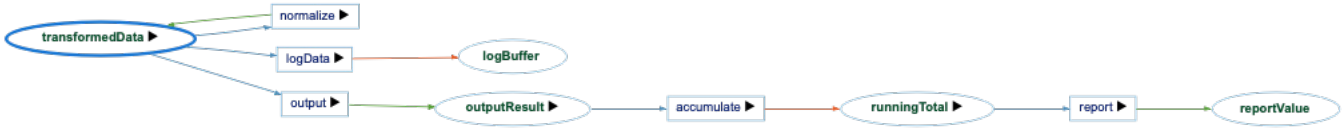
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Data Flow Out

Data Flow Out - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Data Flow Out

Data Flow Out & Data Flow In

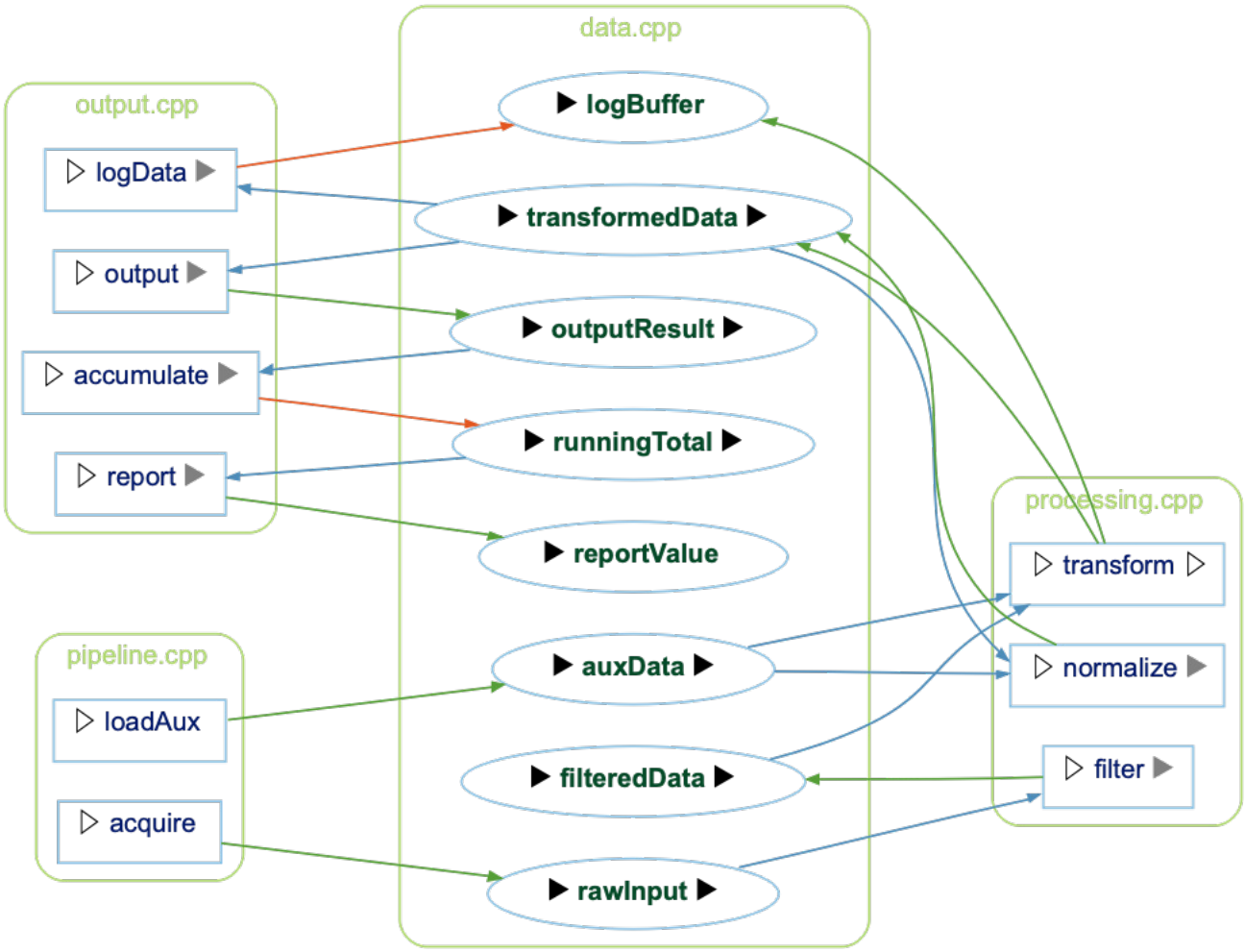
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Classes, Files
Variants: Butterfly, Data Flow In, Data Flow Out, Internal

Show Data Flow Out and Data Flow In relationships for a class, file, or architecture

See Data Flow Out and Data Flow In graphs for details

Data Flow Out & Data Flow In - Butterfly

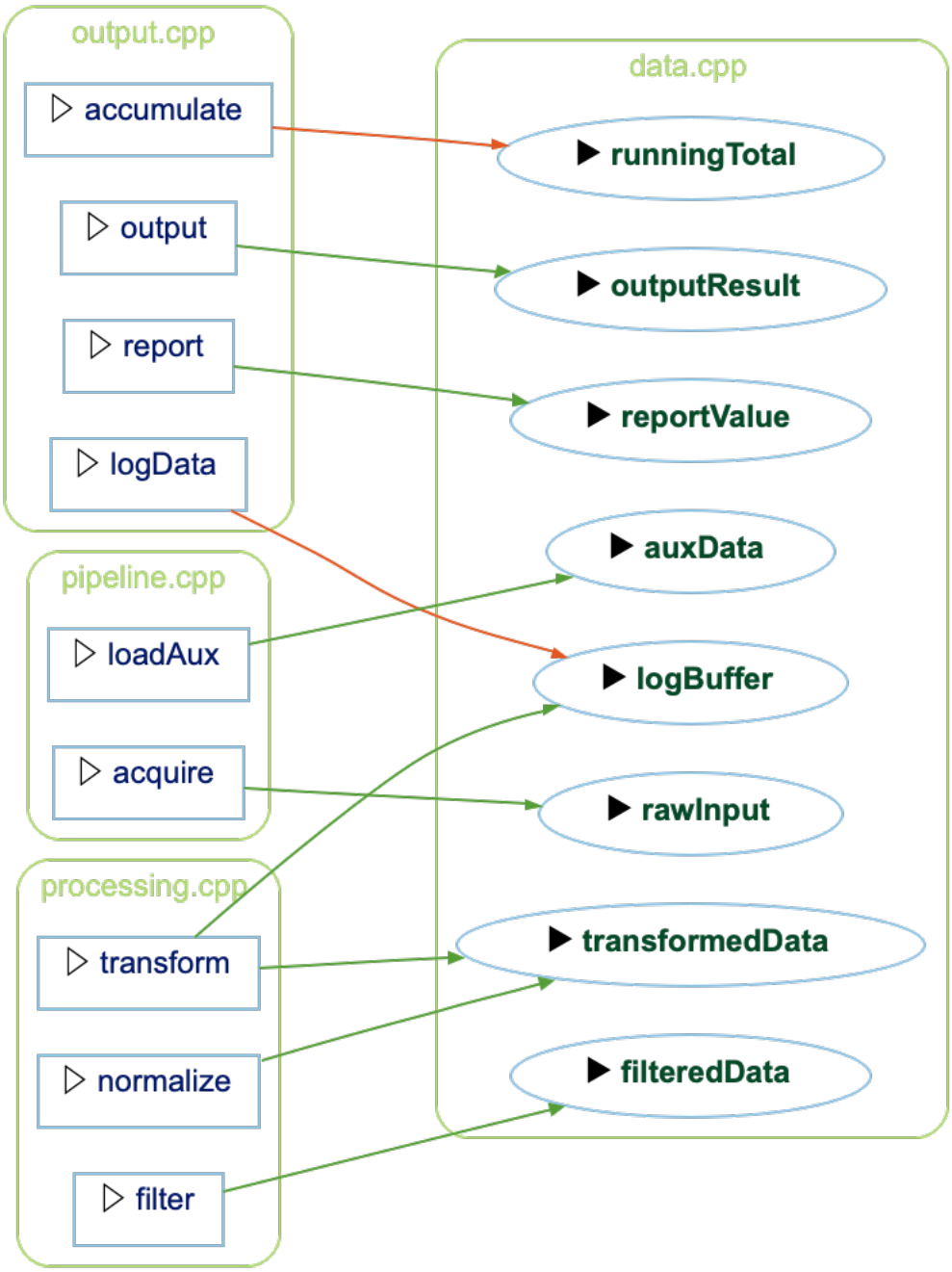
Show both Data Flow Out relationships from and Data Flow In relationships to entities in a class, file, or architecture



Back to Data Flow Out & Data Flow In

Data Flow Out & Data Flow In - Data Flow In

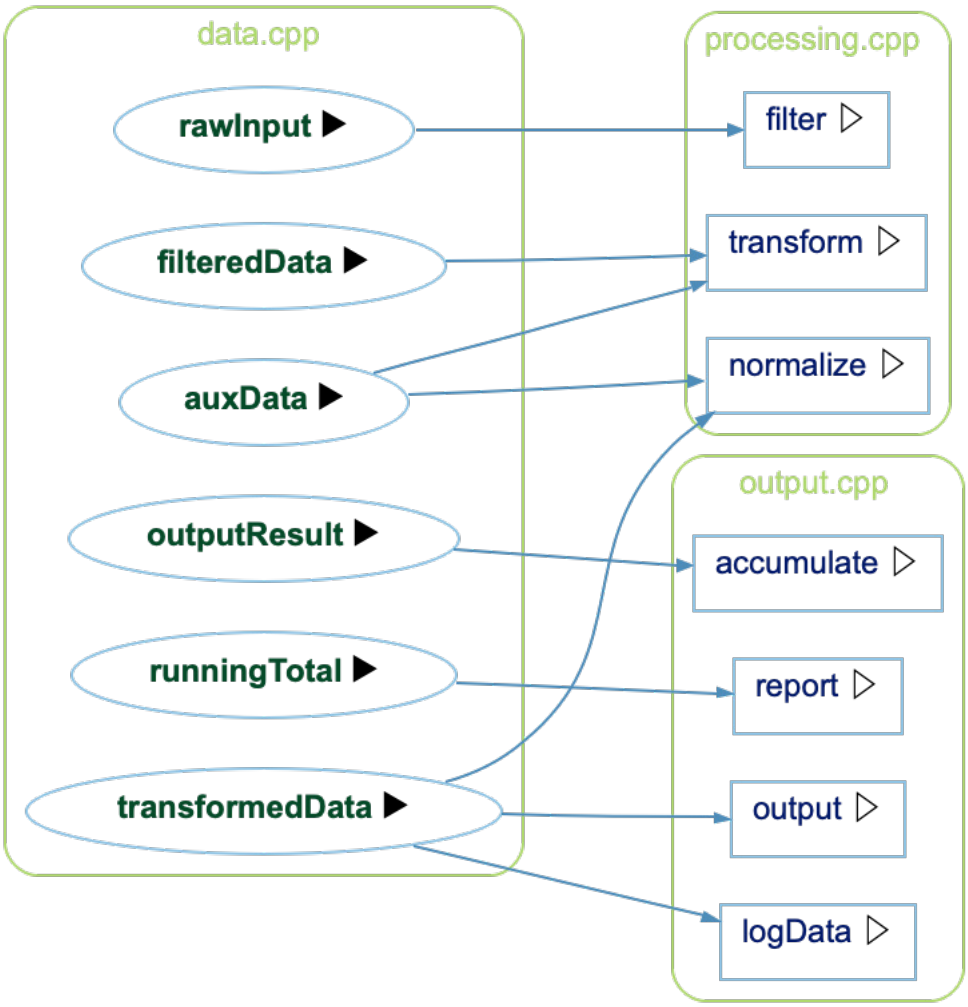
Show Data Flow In relationships to entities in a class, file, or architecture



Back to Data Flow Out & Data Flow In

Data Flow Out & Data Flow In - Data Flow Out

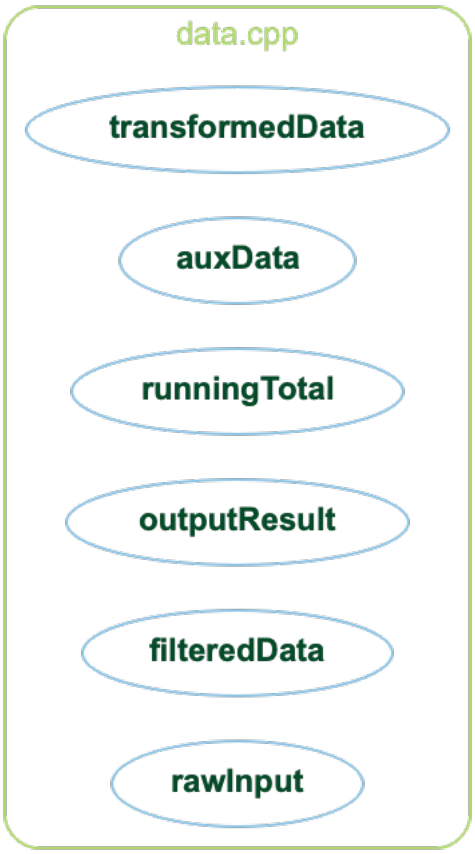
Show Data Flow Out relationships from entities in a class, file, or architecture



Back to Data Flow Out & Data Flow In

Data Flow Out & Data Flow In - Internal

Show Data Flow Out relationships between entities in a class, file, or architecture



Back to Data Flow Out & Data Flow In

Data Members

Languages: C#, C++, Fortran
Targets: Classes, Objects, Types
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the data members nested inside a selected type, down to their primitive fields.

For a selected class, struct, union, or variable, this shows an edge to each of its data members, then continues into any member whose type is itself a class, struct, or union, recursing into that type's own members in turn. Base class members aren't included, since inheritance isn't followed; a plain type alias (typedef) is skipped in favor of the underlying type it names.

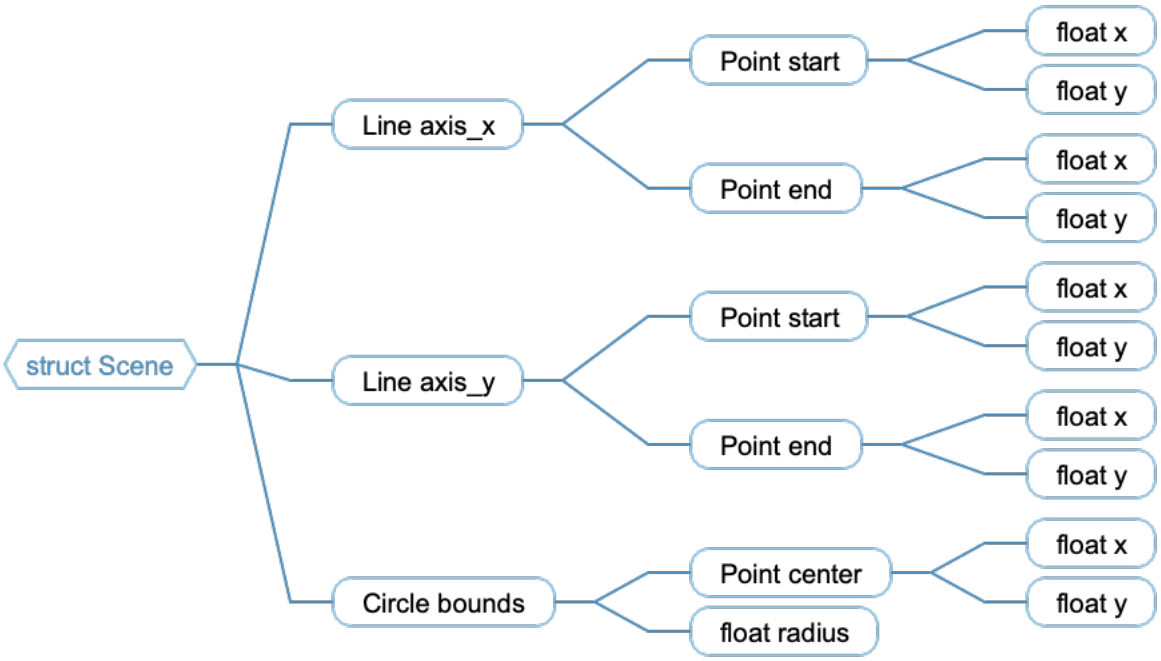
Rooted at Scene below, the graph shows its members axis_x and axis_y (each a Line) and bounds (a Circle). Line continues into its own members start and end (each a Point), and Circle continues into center (a Point) and radius (a plain float). Each Point continues into its primitive members x and y.

See the following code and corresponding graph

```
1 struct Point {
2     float x;
3     float y;
4 };
5
6 struct Line {
7     Point start;
8     Point end;
9 };
10
11 struct Circle {
12     Point center;
13     float radius;
14 };
15
16 struct Scene {
17     Line   axis_x;
18     Line   axis_y;
19     Circle bounds;
20 };
```

Data Members - Classic

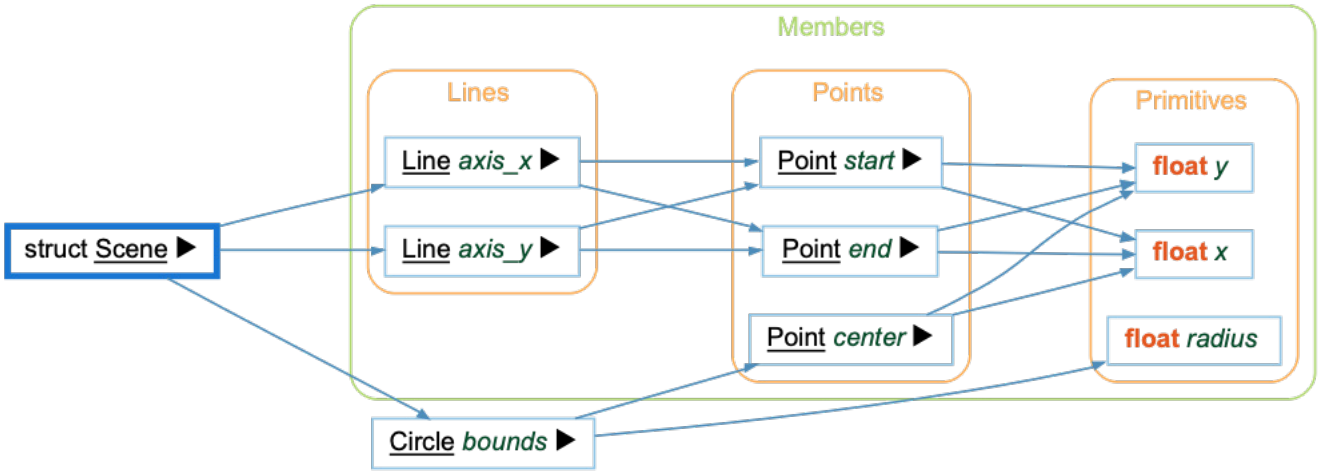
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Data Members

Data Members - Cluster

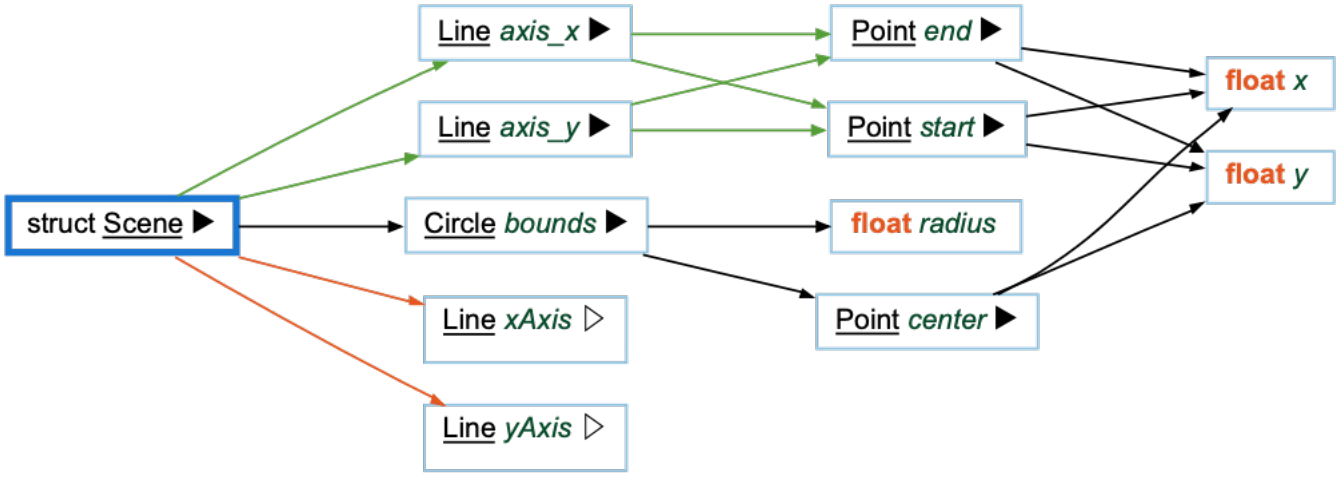
The Cluster variant groups entities by their containing class/file/architecture.



Back to Data Members

Data Members - Compare

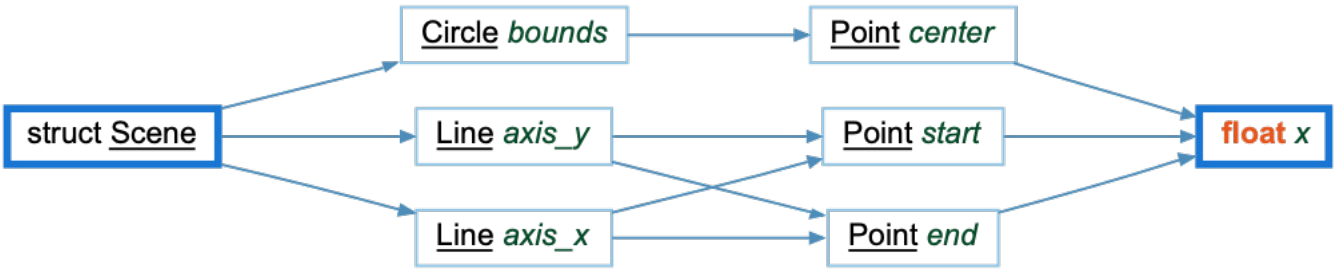
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Data Members

Data Members - Relationship

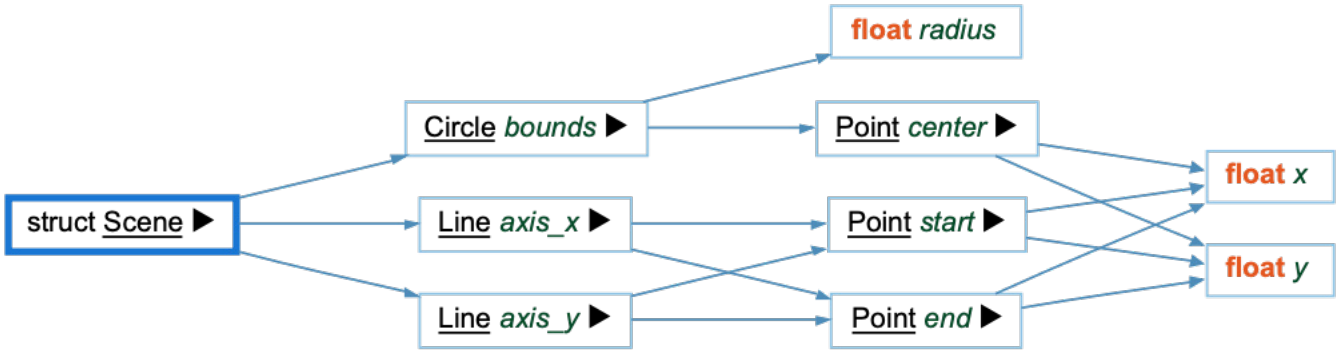
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Data Members

Data Members - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Data Members

Declaration

Languages: Ada, Assembly, C#, C++, Fortran, Java, Jovial, Pascal, Python, Web
Targets: Classes, Files, Functions, Modules, Objects, Packages, Record Types, Subprograms, Types
Variants: Ada, Assembly File, C Class, C Code File, C Header File, C Namespace, C# File, C# Type, Enum, Fortran Common, Fortran Derived Type, Fortran File, Fortran Module, Fortran Subroutine, Function, Java Class, Java File, Jovial File, Pascal Class, Pascal CompUnit Unit, Pascal File Include, Pascal Sql Table, Python Class, Python File, Web Class, Web File

Declaration - Ada

Show a diagram of a selected Ada unit's own declarations, plus (depending on the unit's kind and which options are enabled) the units it withs, is withed by, calls, and is called by.

The main box lists what's declared directly inside the selected unit: subprograms, nested packages, task types, and protected types on the left; types, constants,

exceptions, and objects on the right. What actually shows up depends on the selected unit's own kind — a package can contain any of these, while a task type or protected type can only contain entries, subprograms, and components.

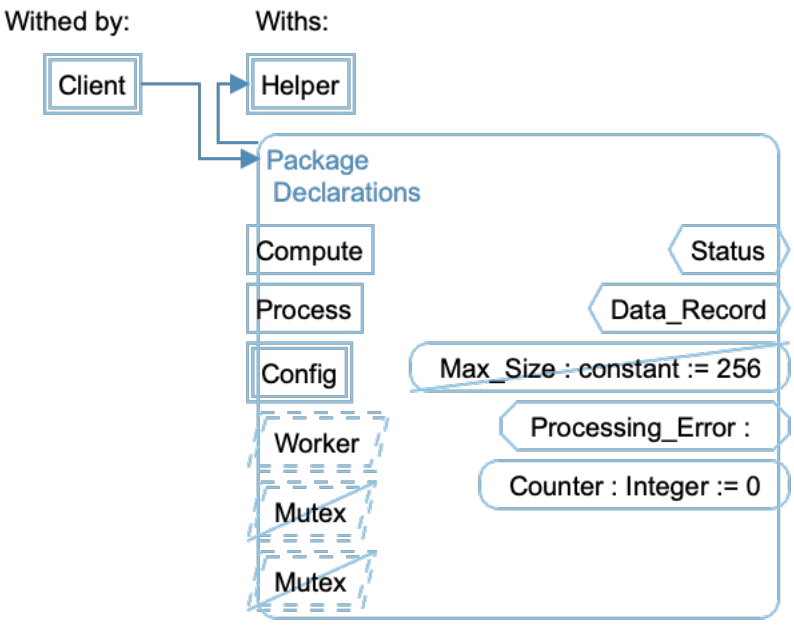
Rooted at the package Declarations below, the left side of the box shows the function Compute, the procedure Process, the nested package Config, the task type Worker, and the protected type Mutex; the right side shows the enumeration type Status, the record type Data_Record, the constant Max_Size, the exception Processing_Error, and the object Counter. Because Declarations withs Helper and is in turn withed by Client, both appear as separate boxes outside the main one.

See the following code and corresponding graph

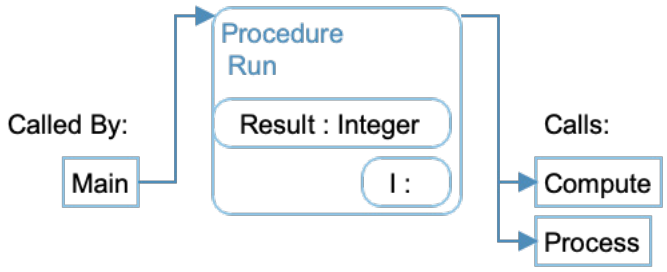
```
1 -- declarations.ads
2 with Helper;
3
4 package Declarations is
5
6    function  Compute (X : Integer) return Integer;
7    procedure Process (Data : in out Integer);
8
9    package Config is
10       Default_Value : constant := 0;
11    end Config;
12
13    task type Worker is
14       entry Start (Id : Integer);
15       entry Stop;
16    end Worker;
17
18    protected type Mutex is
19       entry     Acquire;
20       procedure Release;
21       function  Is_Locked return Boolean;
22    private
23       Locked : Boolean := False;
24    end Mutex;
25
26    type Status is (Pending, Active, Done, Failed);
27
28    type Data_Record is record
29       Value : Integer;
30       Valid : Boolean;
31    end record;
32
33    Max_Size         : constant := 256;
34    Processing_Error : exception;
35    Counter          : Integer := 0;
36
37 end Declarations;
38
```

```
39 -- declarations.adb
40 package body Declarations is
41
42    function Compute (X : Integer) return Integer is
43    begin
44       Helper.Log ("Computing");
45       Counter := Counter + 1;
46       return X * 2;
47    end Compute;
48
49    procedure Process (Data : in out Integer) is
50    begin
51       Helper.Log ("Processing");
52       if Data > Max_Size then
53          raise Processing_Error;
54       end if;
55       Data := Data mod Max_Size;
56    end Process;
57
58    package body Config is
59    end Config;
60
61    task body Worker is
62    begin
63       loop
64          select
65             accept Start (Id : Integer) do
66                null;
67             end Start;
68          or
69             accept Stop;
70             exit;
71          end select;
72       end loop;
73    end Worker;
74
75    protected body Mutex is
76       entry Acquire when not Locked is
77       begin
78          Locked := True;
79       end Acquire;
80
81       procedure Release is
82       begin
83          Locked := False;
84       end Release;
85
86       function Is_Locked return Boolean is
87       begin
88          return Locked;
89       end Is_Locked;
90    end Mutex;
91
```

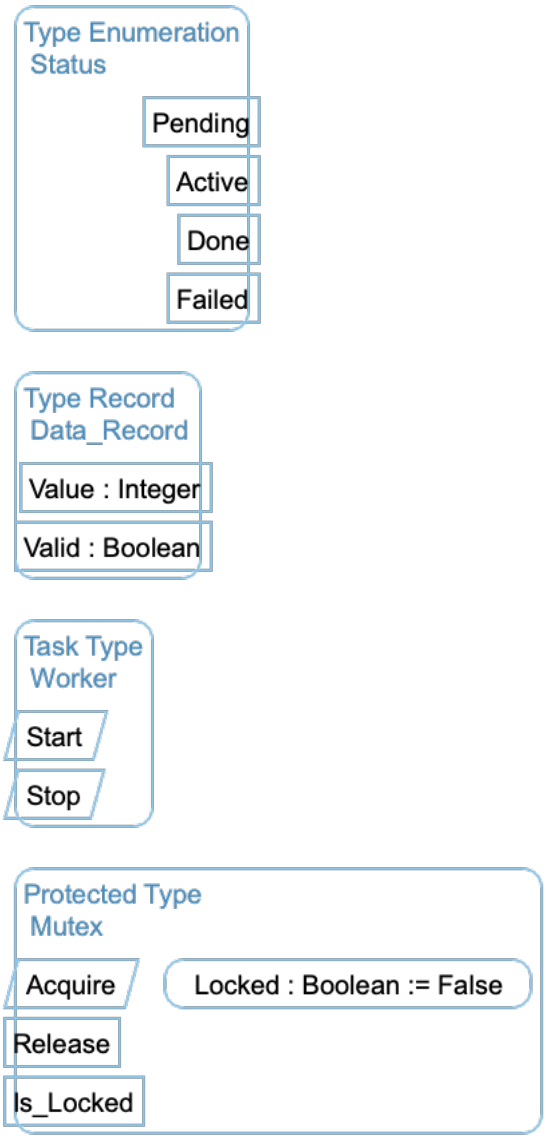
```
92 end Declarations;
93
94 -- helper.ads
95 package Helper is
96    procedure Log (Message : String);
97 end Helper;
98
99 -- client.ads
100 with Declarations;
101
102 package Client is
103    procedure Run (Count : Integer; Label : String);
104 end Client;
105
106 -- client.adb
107 package body Client is
108
109    procedure Run (Count : Integer; Label : String) is
110       Result : Integer;
111    begin
112       for I in 1 .. Count loop
113          Result := Declarations.Compute (I);
114          Declarations.Process (Result);
115       end loop;
116    end Run;
117
118 end Client;
119
120 -- main.adb
121 with Client;
122
123 procedure Main is
124    procedure Run (Count : Integer; Label : String) renames Client.Run;
125 begin
126    Run (10, "test");
127 end Main;
```



For a subprogram, entry, task type, or protected type, the same box can additionally show its parameters and local objects, the units that call it, and the units it calls. Rooted at the procedure Client.Run below, its local object Result and the loop parameter I appear inside the box, Main appears under Called By, and Compute/ Process appear under Calls.



The same box format applies when an enumeration type, record type, task type, or protected type is selected directly, rather than reached through a containing package: an enumeration type lists its literals, a record type lists its components, a task type lists its entries, and a protected type lists its entries, subprograms, and private components.



Back to Declaration

Declaration - Assembly File

Show a diagram of a selected assembly file's own declarations, plus what includes it, what it includes, and what it calls.

The main box lists the macros, symbols and labels, and constants and variables defined directly in the file.

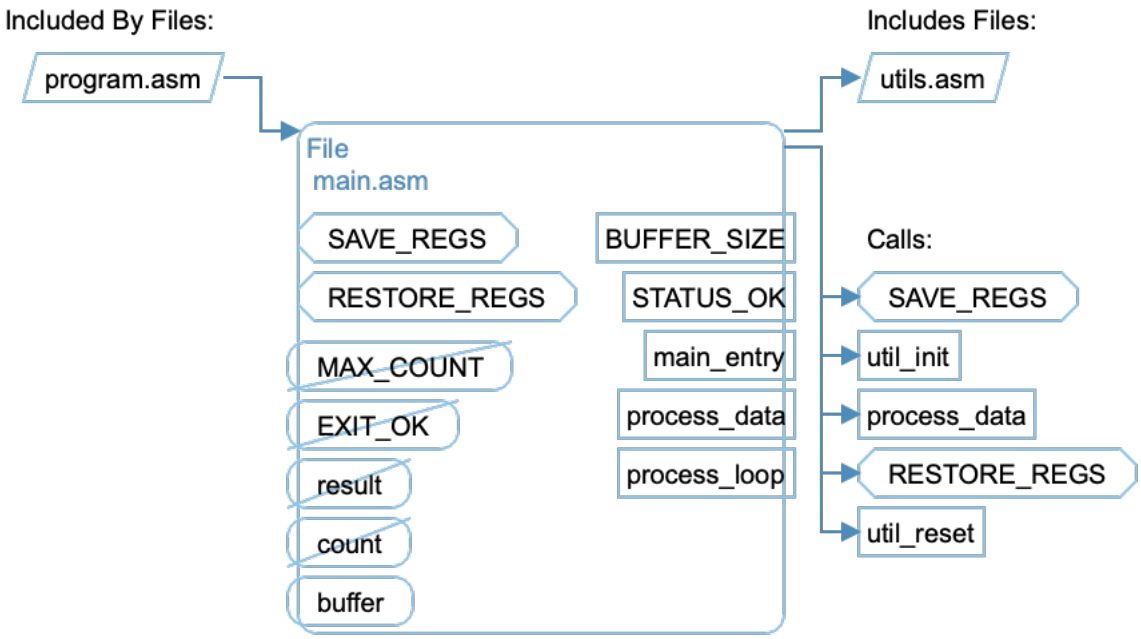
Rooted at main.asm below, the box shows the macros SAVE_REGS and RESTORE_REGS; the symbols BUFFER_SIZE and STATUS_OK and the labels main_entry, process_data, and process_loop; and the constants and variables MAX_COUNT, EXIT_OK, result, count, and buffer. Included By Files shows program.asm, which includes main.asm; Includes Files shows utils.asm, which main.asm includes in turn. Calls shows everything main_entry and process_data

invoke: the SAVE_REGS and RESTORE_REGS macros, and util_init/util_reset from utils.asm.

See the following code and corresponding graph

```
        INCLUDE "utils.asm"

* Macros
SAVE_REGS MACRO
        MOVEM.L D0-D2/A0-A1,-(A7)
        ENDM

RESTORE_REGS MACRO
        MOVEM.L (A7)+,D0-D2/A0-A1
        ENDM

* Symbols
BUFFER_SIZE EQU 256
STATUS_OK   EQU 0

        SECTION DATA

* Constants and variables
MAX_COUNT DC.W 10
EXIT_OK   DC.W 0
result    DC.L 0
count     DC.L 0
buffer    DS.B BUFFER_SIZE

        SECTION CODE

* Labels
main_entry
        SAVE_REGS
        JSR util_init
        JSR process_data
        RESTORE_REGS
        RTS

process_data
        MOVE.W MAX_COUNT,D0
process_loop
        ADDQ.L #1,result
        ADDQ.L #1,count
        DBRA D0,process_loop
```

```
    JSR util_reset
    RTS
```



Back to Declaration

Declaration - C Class

Show a diagram of a selected C/C++ class, struct, or union's own members, plus its base and derived classes.

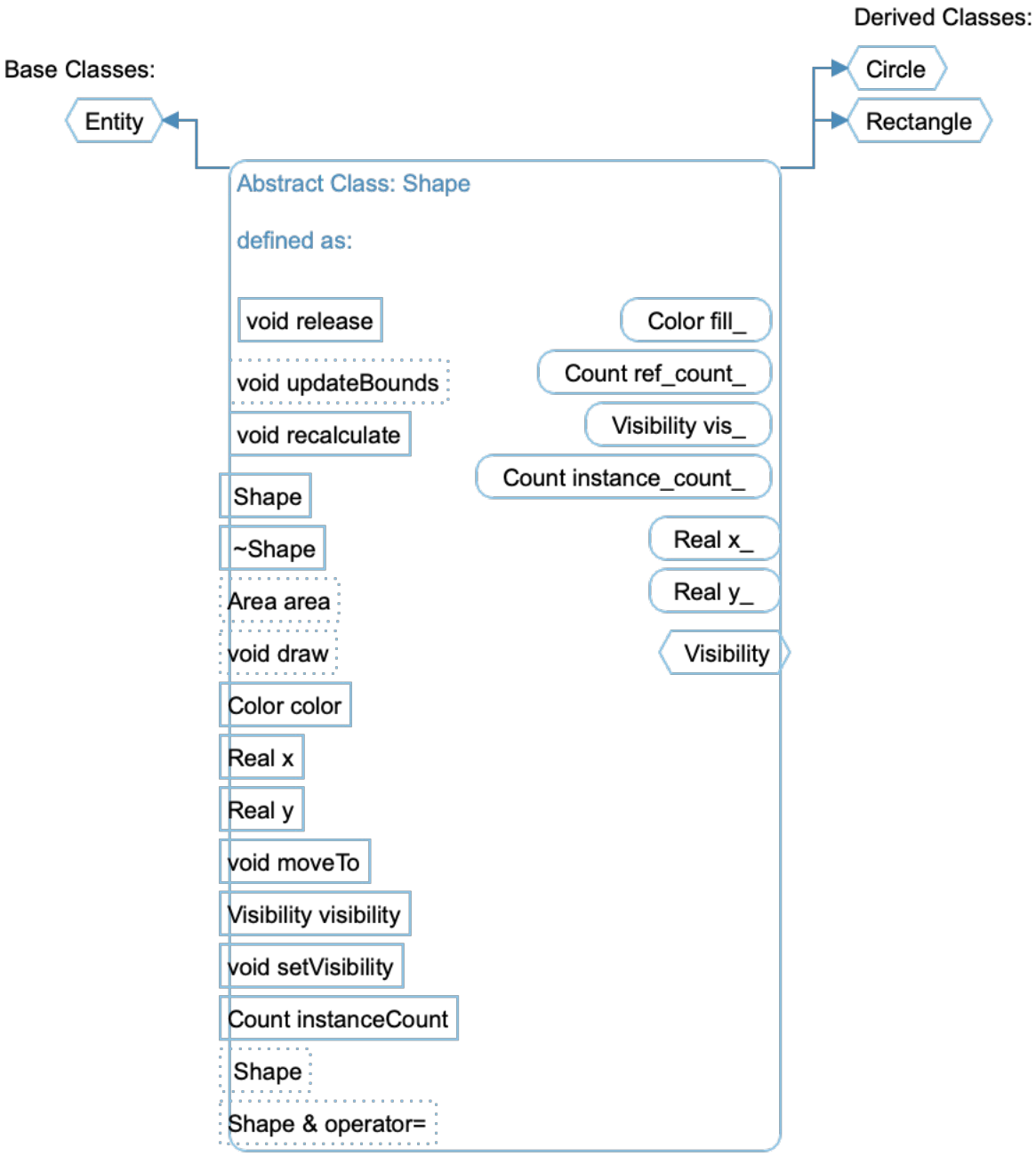
The main box lists private members at the top, protected members in the middle, and public members at the bottom, with functions on the left and data on the right; a nested enum is treated as a public member. Enabling Base Classes and Derived Classes adds the class's immediate ancestors and descendants outside the main box.

Rooted at the abstract class Shape below (abstract because it declares the pure virtual functions area and draw), the box shows the private function release and private data fill_, ref_count_, vis_, and instance_count_; the protected functions updateBounds and recalculate and protected data x_ and y_; and the public constructor, destructor, area, draw, color, x, y, moveTo, visibility, setVisibility, the static instanceCount, the assignment operator, and the nested enum Visibility. Base Classes shows Entity; Derived Classes shows Circle and Rectangle.

See the following code and corresponding graph

```
1 class Entity {
```

```
2 private:
3     static Count next_id_;
4     void assignId();
5 protected:
6     int id_;
7     virtual void initialize() = 0;
8 public:
9     Entity();
10     virtual ~Entity();
11     int id() const;
12 };
13
14 class Shape : public Entity {
15 private:
16     Color     fill_;
17     Count     ref_count_;
18     void      release();
19 protected:
20     Real               x_, y_;
21     virtual void       updateBounds() = 0;
22     virtual void       recalculate(Real tolerance);
23 public:
24     enum Visibility { Visible, Hidden, Clipped };
25
26     Shape(Real x, Real y, Color c);
27     virtual ~Shape();
28     virtual Area  area()  const = 0;
29     virtual void  draw()  const = 0;
30     Color         color() const;
31     Real          x()     const;
32     Real          y()     const;
33     void          moveTo(Real nx, Real ny);
34     Visibility    visibility() const;
35     void          setVisibility(Visibility v);
36
37     static Count  instanceCount();
38
39 private:
40     Visibility vis_;
41     static Count instance_count_;
42 };
43
44 class Circle : public Shape {
45     // ...
46 };
47
48 class Rectangle : public Shape {
49     // ...
50 };
```



Back to Declaration

Declaration - C Code File

Show a diagram of a selected C/C++ source file's own definitions, plus the header files it includes.

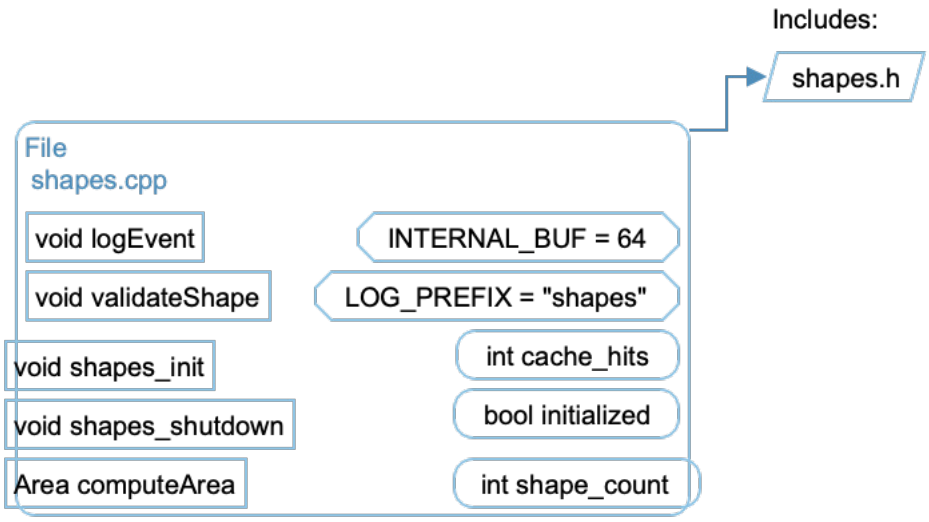
The main box lists the file's non-static functions and global objects; enabling the Static option adds its static functions, static objects, macros, and other file-local types too.

Rooted at shapes.cpp below, the box shows the functions shapes_init,

shapes_shutdown, and computeArea and the global object shape_count. With Static enabled, it also shows the static functions logEvent and validateShape, the static objects cache_hits and initialized, and the macros INTERNAL_BUF and LOG_PREFIX. Includes shows shapes.h, which the file includes.

See the following code and corresponding graph

```
1 #include "shapes.h"
2
3 #define INTERNAL_BUF   64
4 #define LOG_PREFIX     "shapes"
5
6 static int   cache_hits   = 0;
7 static bool  initialized  = false;
8
9 static void logEvent(const char* msg, int level) {
10     (void)msg; (void)level;
11     ++cache_hits;
12 }
13
14 static void validateShape(const Shape* s) {
15     (void)s;
16 }
17
18 int shape_count = 0;
19
20 void shapes_init() {
21     if (!initialized) {
22         logEvent("shapes_init", 1);
23         initialized = true;
24     }
25 }
26
27 void shapes_shutdown() {
28     logEvent("shapes_shutdown", 1);
29     initialized = false;
30     shape_count = 0;
31 }
32
33 Area computeArea(const void* shape, Count iterations, Real scale) {
34     (void)iterations;
35     Area total = 0.0;
36     const Shape* s = static_cast(shape);
37     validateShape(s);
38     if (s) {
39         total = s->area() * scale;
40         logEvent("computeArea", 2);
41     }
42     return total;
43 }
```



Back to Declaration

Declaration - C Header File

Show a diagram of a selected C/C++ header file's own declarations, plus what includes it and what it includes.

The main box lists the file's macros and non-static functions on the left and its objects and other types on the right — including anything declared inside a namespace in the file, not just what's declared at file scope directly. Enabling External Functions adds a separate list of the functions the file declares but doesn't define.

Rooted at shapes.h below, the left side of the box shows the macros SHAPES_H, PI, MAX_SHAPES, and DEGREES_TO_RADIANS and the inline function square; the right side shows the type aliases Area and Angle, the enum Color, the classes Entity, Shape, and Circle, the type alias ShapeAlias (for Shape), the class Rectangle, the struct Point, the class BoundingBox, and the type aliases Coordinate and Index — even though several of these are declared inside the namespace Geometry, not directly in the file. External Functions lists shapes_init, shapes_shutdown, computeArea, distance, angle, trace, and assertValid. Included By shows main.cpp and shapes.cpp; Includes shows platform.h.

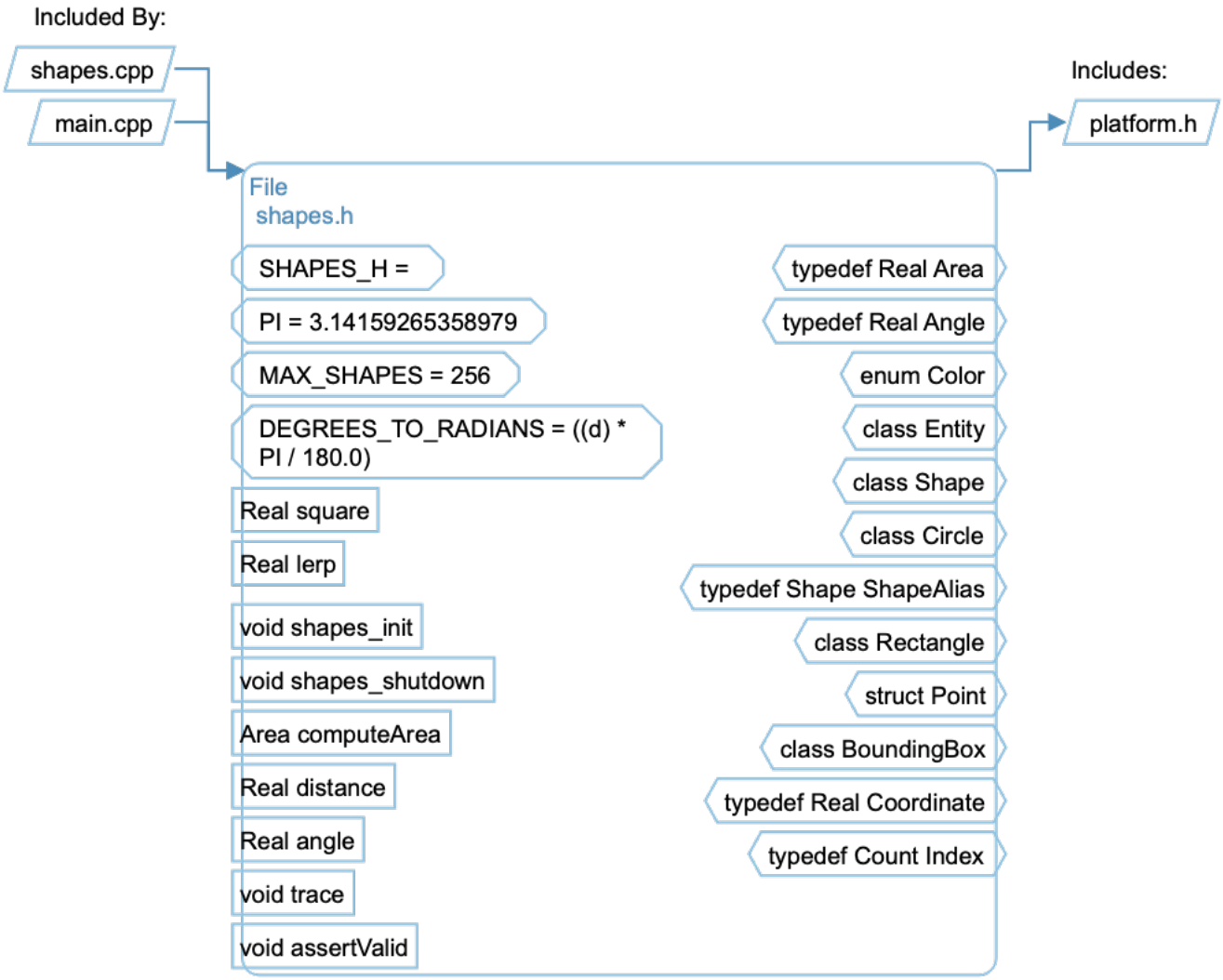
See the following code and corresponding graph

```
1 #ifndef SHAPES_H
2 #define SHAPES_H
3
4 #include "platform.h"
5
6 #define PI                    3.14159265358979
7 #define MAX_SHAPES            256
8 #define DEGREES_TO_RADIANS(d) ((d) * PI / 180.0)
```

```
9
10 inline Real square(Real v) { return v * v; }
11
12 extern int shape_count;
13
14 typedef Real Area;
15 typedef Real Angle;
16
17 enum Color { Red, Green, Blue, Alpha };
18
19 void shapes_init();
20 void shapes_shutdown();
21 Area computeArea(const void* shape, Count iterations, Real scale);
22
23 namespace Geometry {
24     namespace Detail {
25         void trace(const char* msg, int level);
26         void assertValid(bool condition, const char* context);
27     }
28
29     struct Point {
30         Real x;
31         Real y;
32     };
33
34     class BoundingBox {
35     public:
36         BoundingBox(Real x, Real y, Real w, Real h);
37         bool contains(Point p) const;
38         Real width() const;
39         Real height() const;
40     private:
41         Real x_, y_, w_, h_;
42     };
43
44     typedef Real Coordinate;
45     typedef Count Index;
46
47     Real distance(Point a, Point b);
48     Real angle(Point a, Point b);
49     inline Real lerp(Real a, Real b, Real t)
50         { return a + (b - a) * t; }
51     static inline Real clamp(Real v, Real lo, Real hi)
52         { return v  hi ? hi : v); }
53
54     extern Real   scale_factor;
55     extern Count  render_count;
56 }
57
58 class Entity {
59 private:
60     static Count next_id_;
61     void assignId();
```

```
62 protected:
63     int id_;
64     virtual void initialize() = 0;
65 public:
66     Entity();
67     virtual ~Entity();
68     int id() const;
69 };
70
71 class Shape : public Entity {
72 private:
73     Color     fill_;
74     Count     ref_count_;
75     void      release();
76 protected:
77     Real               x_, y_;
78     virtual void       updateBounds() = 0;
79     virtual void       recalculate(Real tolerance);
80 public:
81     enum Visibility { Visible, Hidden, Clipped };
82
83     Shape(Real x, Real y, Color c);
84     virtual ~Shape();
85     virtual Area  area()  const = 0;
86     virtual void  draw()  const = 0;
87     Color         color() const;
88     Real          x()     const;
89     Real          y()     const;
90     void          moveTo(Real nx, Real ny);
91     Visibility    visibility() const;
92     void          setVisibility(Visibility v);
93
94     static Count  instanceCount();
95
96 private:
97     Visibility vis_;
98     static Count instance_count_;
99 };
100
101 class Circle : public Shape {
102 private:
103     Real  radius_;
104     Count segments_;
105 protected:
106     void updateBounds() override;
107     void recalculate(Real tolerance) override;
108 public:
109     Circle(Real x, Real y, Real r, Color c);
110     ~Circle() override;
111     Area area()  const override;
112     void draw()  const override;
113     Real radius()      const;
114     void setRadius(Real r);
```

```
115     Geometry::BoundingBox bounds() const;
116 protected:
117     void initialize() override;
118 };
119 
120 typedef Shape ShapeAlias;
121 
122 class Rectangle : public Shape {
123 private:
124     Real width_, height_;
125 protected:
126     void updateBounds() override;
127     void recalculate(Real tolerance) override;
128 public:
129     Rectangle(Real x, Real y, Real w, Real h, Color c);
130     ~Rectangle() override;
131     Area area()   const override;
132     void draw()   const override;
133     Real width()  const;
134     Real height() const;
135 protected:
136     void initialize() override;
137 };
138 
139 #endif
```



Back to Declaration

Declaration - C Namespace

Show a diagram of a selected C++ namespace's own declarations.

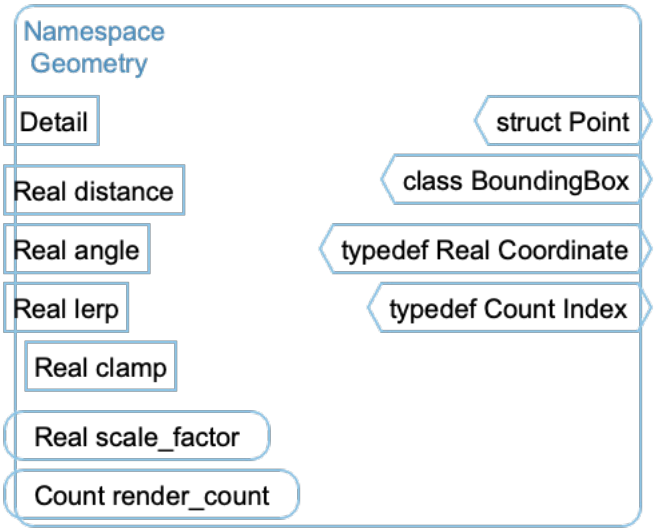
The main box lists nested namespaces at the top left, classes and structs at the top right, other types at the middle right, functions at the middle left, and objects at the bottom left.

Rooted at the namespace Geometry below, the box shows the nested namespace Detail; the struct Point and class BoundingBox; the type aliases Coordinate and Index; the functions distance, angle, lerp, and clamp; and the objects scale_factor and render_count.

See the following code and corresponding graph

```
1 namespace Geometry {
2
```

```
3     namespace Detail {
4         void trace(const char* msg, int level);
5         void assertValid(bool condition, const char* context);
6     }
7
8     struct Point {
9         Real x;
10         Real y;
11     };
12
13     class BoundingBox {
14     public:
15         BoundingBox(Real x, Real y, Real w, Real h);
16         bool contains(Point p) const;
17         Real width() const;
18         Real height() const;
19     private:
20         Real x_, y_, w_, h_;
21     };
22
23     typedef Real Coordinate;
24     typedef Count Index;
25
26     Real distance(Point a, Point b);
27     Real angle(Point a, Point b);
28     inline Real lerp(Real a, Real b, Real t)
29         { return a + (b - a) * t; }
30     static inline Real clamp(Real v, Real lo, Real hi)
31         { return v  hi ? hi : v); }
32
33     extern Real   scale_factor;
34     extern Count  render_count;
35
36 } // namespace Geometry
```



Back to Declaration

Declaration - C# File

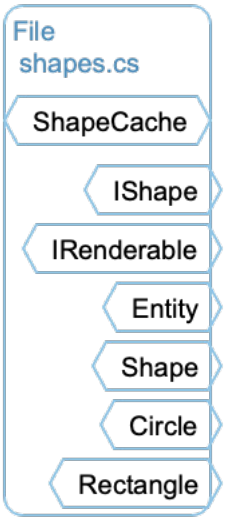
Show a diagram of a selected C# file's own type declarations.

The main box lists the types declared directly in the file, grouped by accessibility: private types at the top, protected and internal types in the middle, and public types at the bottom.

Rooted at shapes.cs below, the box shows the public types IShape, IRenderable, Entity, Shape, Circle, and Rectangle, and the internal type ShapeCache.

See the following code and corresponding graph

```
1 namespace Geometry
2 {
3     public interface IShape { /* ... */ }
4     public interface IRenderable { /* ... */ }
5
6     internal sealed class ShapeCache { /* ... */ }
7
8     public abstract class Entity { /* ... */ }
9
10     public abstract class Shape
11         : Entity, IShape, IRenderable { /* ... */ }
12
13     public sealed class Circle : Shape { /* ... */ }
14
15     public sealed class Rectangle : Shape { /* ... */ }
16 }
```



Back to Declaration

Declaration - C# Type

Show a diagram of a selected C# type's own members, plus its base types, derived types, and interfaces.

This applies to any C# type declaration — class, struct, interface, record, delegate, tuple, or generic type parameter — though a delegate, tuple, or generic type parameter has no members, base types, or interfaces of its own, so the box is mostly empty for those. For a class, struct, interface, or record, the main box lists private members at the top, protected and internal members in the middle, and public members at the bottom, with methods on the left and fields, properties, indexers, events, and nested types on the right; a nested enum is treated as a public member. Enabling Base Classes and Derived Classes adds the type's immediate ancestors and descendants outside the main box; Implements and Implemented By add the interfaces it implements and, for an interface, the types that implement it.

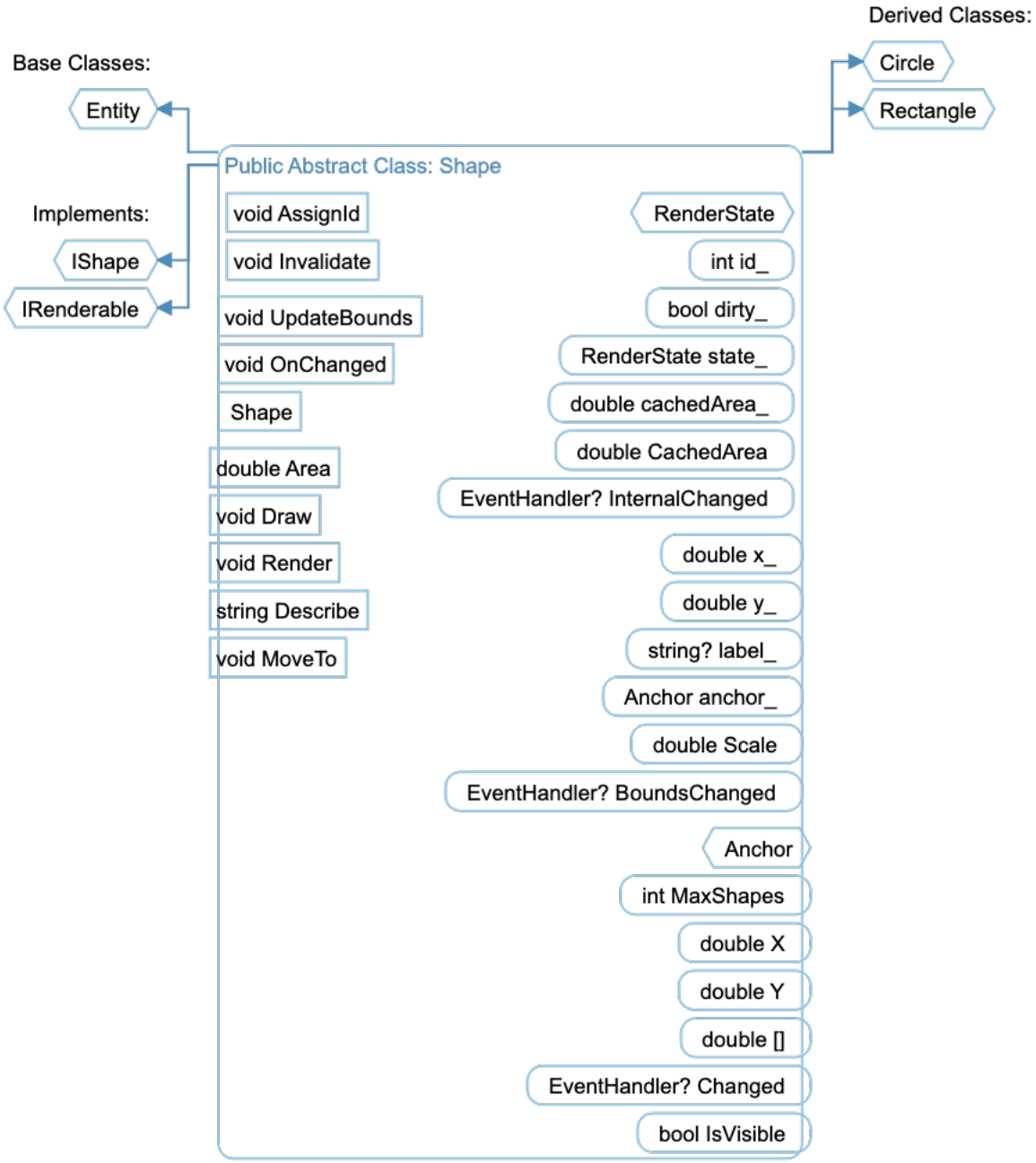
Rooted at the abstract class Shape below, the box shows the private members AssignId and Invalidate, the nested private class RenderState, and the private fields id_, dirty_, state_, and cachedArea_ and property CachedArea; the protected members UpdateBounds and OnChanged and the protected fields x_, y_, label_, and anchor_, property Scale, and event BoundsChanged; and the public members Area, Draw, Render, Describe, and MoveTo, the nested public enum Anchor, the public field MaxShapes, properties X, Y, and IsVisible, the indexer this[int], and the event Changed. Base Classes shows Entity; Derived Classes shows Circle and Rectangle; Implements shows IShape and IRenderable.

See the following code and corresponding graph

```
1 public interface IShape
2 {
3     double Area();
4     void   Draw();
5 }
6
7 public interface IRenderable
8 {
9     void Render(double opacity);
10     bool IsVisible { get; }
11 }
12
13 public abstract class Entity
14 {
15     private static int nextId_ = 0;
16
17     protected int Id { get; }
18
19     protected Entity() { Id = ++nextId_; }
20
21     public abstract string Describe();
```

```
22 }
23
24 public abstract class Shape : Entity, IShape, IRenderable
25 {
26     private class RenderState
27     {
28         public bool   Active;
29         public int    Frame;
30         public double Opacity;
31     }
32
33     public enum Anchor {
34         TopLeft, TopRight, BottomLeft, BottomRight, Center }
35
36     private int         id_;
37     private bool        dirty_;
38     private RenderState state_;
39     private double      cachedArea_;
40
41     private double CachedArea
42     {
43         get => cachedArea_;
44         set { cachedArea_ = value; dirty_ = false; }
45     }
46
47     private event EventHandler? InternalChanged;
48
49     private void AssignId()    { id_ = Id; }
50     private void Invalidate() {
51         dirty_ = true;
52         InternalChanged?.Invoke(this, EventArgs.Empty); }
53
54     protected double  x_;
55     protected double  y_;
56     protected string? label_;
57     protected Anchor  anchor_;
58
59     protected double Scale { get; set; } = 1.0;
60
61     protected event EventHandler? BoundsChanged;
62
63     protected virtual void UpdateBounds()
64         => BoundsChanged?.Invoke(this, EventArgs.Empty);
65     protected virtual void OnChanged() => Invalidate();
66
67     public static int   MaxShapes = 1024;
68
69     public double X
70     {
71         get => x_;
72         set { x_ = value; UpdateBounds(); }
73     }
74
```

```
75      public double Y
76      {
77          get => y_;
78          set { y_ = value; UpdateBounds(); }
79      }
80
81      public double this[int axis]
82      {
83          get => axis == 0 ? x_ : y_;
84          set { if (axis == 0) x_ = value;
85                else y_ = value; UpdateBounds(); }
86      }
87
88      public event EventHandler? Changed;
89
90      public bool IsVisible { get; set; } = true;
91
92      public abstract double Area();
93      public abstract void   Draw();
94      public virtual  void   Render(double opacity) => Draw();
95      public override string Describe()
96          => $"{GetType().Name}({x_},{y_})";
97      public void MoveTo(double nx, double ny)
98      {
99          x_ = nx; y_ = ny; UpdateBounds();
100          Changed?.Invoke(this, EventArgs.Empty);
101      }
102
103      protected Shape(double x, double y)
104      {
105          x_ = x; y_ = y;
106          state_ = new RenderState();
107          anchor_ = Anchor.Center;
108          AssignId();
109      }
110  }
111
112  public sealed class Circle : Shape { /* ... */ }
113  public sealed class Rectangle : Shape { /* ... */ }
```



Back to Declaration

Declaration - Enum

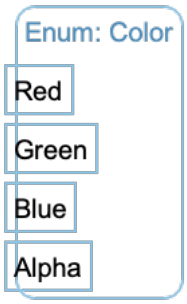
Show a diagram of a selected enum's enumerators.

The main box lists the enum's enumerators.

Rooted at the enum Color below, the box shows its enumerators Red, Green, Blue, and Alpha.

See the following code and corresponding graph

```
1 enum Color { Red, Green, Blue, Alpha };
```



Back to Declaration

Declaration - Fortran Common

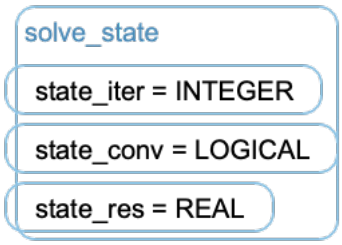
Show a diagram of a selected Fortran common block or datapool's own variables.

The main box lists the variables declared in the common block, along with their types.

Rooted at solve_state below, the box shows its variables state_iter (INTEGER), state_conv (LOGICAL), and state_res (REAL).

See the following code and corresponding graph

```
1 integer :: state_iter
2 logical :: state_conv
3 real    :: state_res
4 common /solve_state/ state_iter, state_conv, state_res
```



Back to Declaration

Declaration - Fortran Derived Type

Show a diagram of a selected Fortran derived type, interface, or pointer's own members.

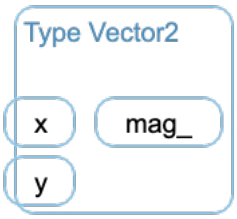
The main box lists the type's public, non-private components and functions on the left and its private ones on the right; type-bound procedures aren't included, only

the type's own data components and any function or subroutine declared directly inside it.

Rooted at the derived type Vector2 below, the box shows its public components x and y on the left and its private component mag_ on the right; its type-bound procedures normalize, length, and compute_mag aren't shown, since they're declared as ordinary functions inside geometry_mod rather than inside Vector2 itself.

See the following code and corresponding graph

```
1 type, public :: Vector2
2     real, public :: x = 0.0
3     real, public :: y = 0.0
4     real, private :: mag_ = 0.0
5 contains
6     procedure, public  :: normalize
7     procedure, public  :: length
8     procedure, private :: compute_mag
9 end type Vector2
```



Back to Declaration

Declaration - Fortran File

Show a diagram of a selected Fortran file's own declarations, plus what includes it.

The main box lists the modules, programs, subroutines, functions, common blocks, and datapools defined directly in the file. Included By lists the subprograms that include the file.

Rooted at geometry.f90 below, the box shows the module geometry_mod, the only thing the file defines.

See the following code and corresponding graph

```
1 module geometry_mod
2     use constants_mod
3     implicit none
4     private
5
```

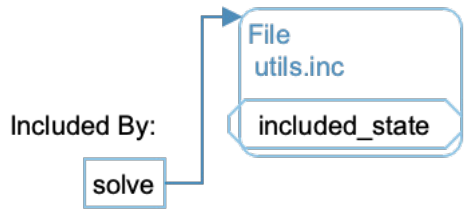
```
    integer, public :: max_vectors = 256

    real,    private :: tolerance_   = 1.0e-6
    logical, private :: initialized_ = .false.

     type, public :: Vector2
         real, public :: x = 0.0
         real, public :: y = 0.0
         real, private :: mag_ = 0.0
     contains
         procedure, public  :: normalize
         procedure, public  :: length
         procedure, private :: compute_mag
     end type Vector2

     type, private :: WorkBuffer
         integer :: size = 0
         real    :: data(64)
     end type WorkBuffer

     interface interpolate
         module procedure lerp_impl
     end interface interpolate

     public :: Vector2, transform, dot, interpolate, max_vectors

contains

     subroutine transform(v, angle)
         type(Vector2), intent(inout) :: v
         real,          intent(in)    :: angle
         real :: cosA, sinA, tmp
         cosA = cos(angle * DEG2RAD)
         sinA = sin(angle * DEG2RAD)
         tmp  = v%x * cosA - v%y * sinA
         v%y  = v%x * sinA + v%y * cosA
         v%x  = tmp
         v%mag_ = 0.0
     end subroutine transform

     real function dot(a, b)
         type(Vector2), intent(in) :: a, b
         dot = a%x * b%x + a%y * b%y
     end function dot

     real function lerp_impl(a, b, t)
         real, intent(in) :: a, b, t
         lerp_impl = a + (b - a) * t
     end function lerp_impl

     real function clamp_val(v, lo, hi)
         real, intent(in) :: v, lo, hi
         clamp_val = min(hi, max(lo, v))
```

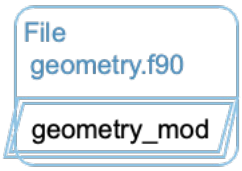
```
59      end function clamp_val
60
61      subroutine normalize(self)
62          class(Vector2), intent(inout) :: self
63          real :: len
64          len = self%compute_mag()
65          if (len > tolerance_) then
66              self%x    = self%x / len
67              self%y    = self%y / len
68              self%mag_ = 1.0
69          end if
70      end subroutine normalize
71
72      real function length(self)
73          class(Vector2), intent(in) :: self
74          length = self%compute_mag()
75      end function length
76
77      real function compute_mag(self)
78          class(Vector2), intent(in) :: self
79          compute_mag = sqrt(self%x**2 + self%y**2)
80      end function compute_mag
81
82  end module geometry_mod
```

A file can also be reached only through an INCLUDE statement rather than compiled directly, as with utils.inc below. Its box shows the common block included_state, which the file defines directly, and Included By shows the subroutine solve, which includes it.

See the following code and corresponding graph

```
1  integer, parameter :: MAX_ITER  = 500
2  real,    parameter :: SMALL_VAL = 1.0e-10
3
4  integer :: included_calls
5  logical :: included_ready
6  common /included_state/ included_calls, included_ready
```





Back to Declaration

Declaration - Fortran Module

Show a diagram of a selected Fortran module's own declarations, plus what uses it and what it uses.

The main box lists what's declared directly in the module: public subprograms and variables on the left, private ones on the right; public derived types and interfaces further down on the left, private ones on the right. Used By lists the programs and subprograms that use the module; Uses lists the other modules it uses.

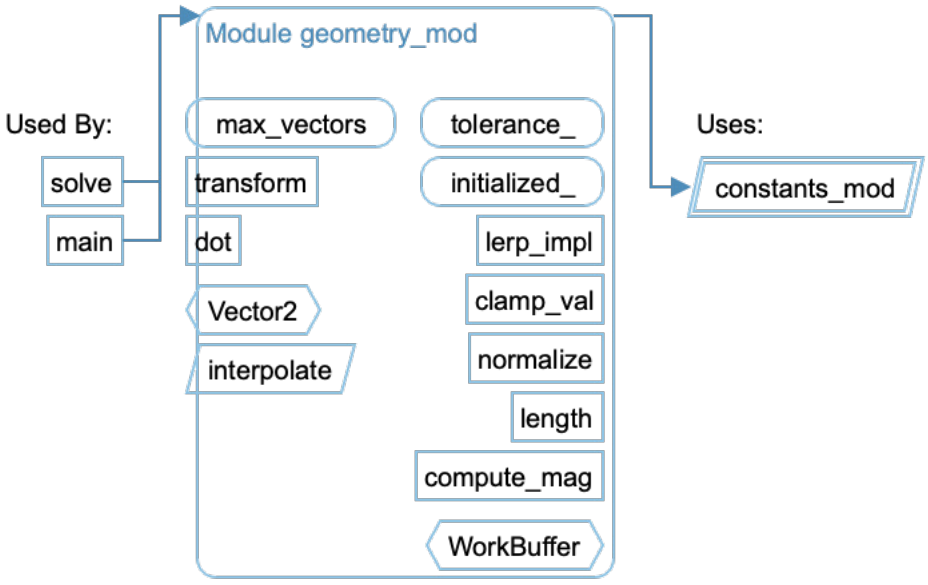
Rooted at geometry_mod below, the box shows the public subroutine transform, function dot, and variable max_vectors; the private functions lerp_impl and clamp_val, variables tolerance_ and initialized_, and the type-bound procedures normalize, length, and compute_mag — private because none of them are named in the module's own public list, even though normalize and length are exposed as public bindings on Vector2. The public derived type Vector2 and interface interpolate appear further down on the left; the private derived type WorkBuffer appears on the right. Used By shows solve and the program main; Uses shows constants_mod.

See the following code and corresponding graph

```
1 module constants_mod
2     implicit none
3     real, parameter :: PI      = 3.14159265358979
4     real, parameter :: TWO_PI  = 6.28318530717959
5     real, parameter :: DEG2RAD = PI / 180.0
6     real, parameter :: RAD2DEG = 180.0 / PI
7 end module constants_mod
8
9 module geometry_mod
10     use constants_mod
11     implicit none
12     private
13
14     integer, public :: max_vectors = 256
15
16     real,    private :: tolerance_   = 1.0e-6
17     logical, private :: initialized_ = .false.
18
19     type, public :: Vector2
20         real, public :: x = 0.0
```

```
21          real, public :: y = 0.0
22          real, private :: mag_ = 0.0
23      contains
24          procedure, public  :: normalize
25          procedure, public  :: length
26          procedure, private :: compute_mag
27      end type Vector2
28
29      type, private :: WorkBuffer
30          integer :: size = 0
31          real    :: data(64)
32      end type WorkBuffer
33
34      interface interpolate
35          module procedure lerp_impl
36      end interface interpolate
37
38      public :: Vector2, transform, dot, interpolate, max_vectors
39
40  contains
41
42      subroutine transform(v, angle)
43          type(Vector2), intent(inout) :: v
44          real,          intent(in)    :: angle
45          real :: cosA, sinA, tmp
46          cosA = cos(angle * DEG2RAD)
47          sinA = sin(angle * DEG2RAD)
48          tmp  = v%x * cosA - v%y * sinA
49          v%y  = v%x * sinA + v%y * cosA
50          v%x  = tmp
51          v%mag_ = 0.0
52      end subroutine transform
53
54      real function dot(a, b)
55          type(Vector2), intent(in) :: a, b
56          dot = a%x * b%x + a%y * b%y
57      end function dot
58
59      real function lerp_impl(a, b, t)
60          real, intent(in) :: a, b, t
61          lerp_impl = a + (b - a) * t
62      end function lerp_impl
63
64      real function clamp_val(v, lo, hi)
65          real, intent(in) :: v, lo, hi
66          clamp_val = min(hi, max(lo, v))
67      end function clamp_val
68
69      subroutine normalize(self)
70          class(Vector2), intent(inout) :: self
71          real :: len
72          len = self%compute_mag()
73          if (len > tolerance_) then
```

```
74              self%x    = self%x / len
75              self%y    = self%y / len
76              self%mag_ = 1.0
77          end if
78      end subroutine normalize
79
80      real function length(self)
81          class(Vector2), intent(in) :: self
82          length = self%compute_mag()
83      end function length
84
85      real function compute_mag(self)
86          class(Vector2), intent(in) :: self
87          compute_mag = sqrt(self%x**2 + self%y**2)
88      end function compute_mag
89
90  end module geometry_mod
91
92  ! solver.f90
93  subroutine solve(n, values, result, tol)
94      use geometry_mod, only: dot, Vector2
95      ! ...
96  end subroutine solve
97
98  ! program.f90
99  program main
100     use geometry_mod
101     ! ...
102 end program main
```



Back to Declaration

Declaration - Fortran Subroutine

Show a diagram of a selected Fortran subprogram's parameters, plus what it includes, uses, calls, and is called by.

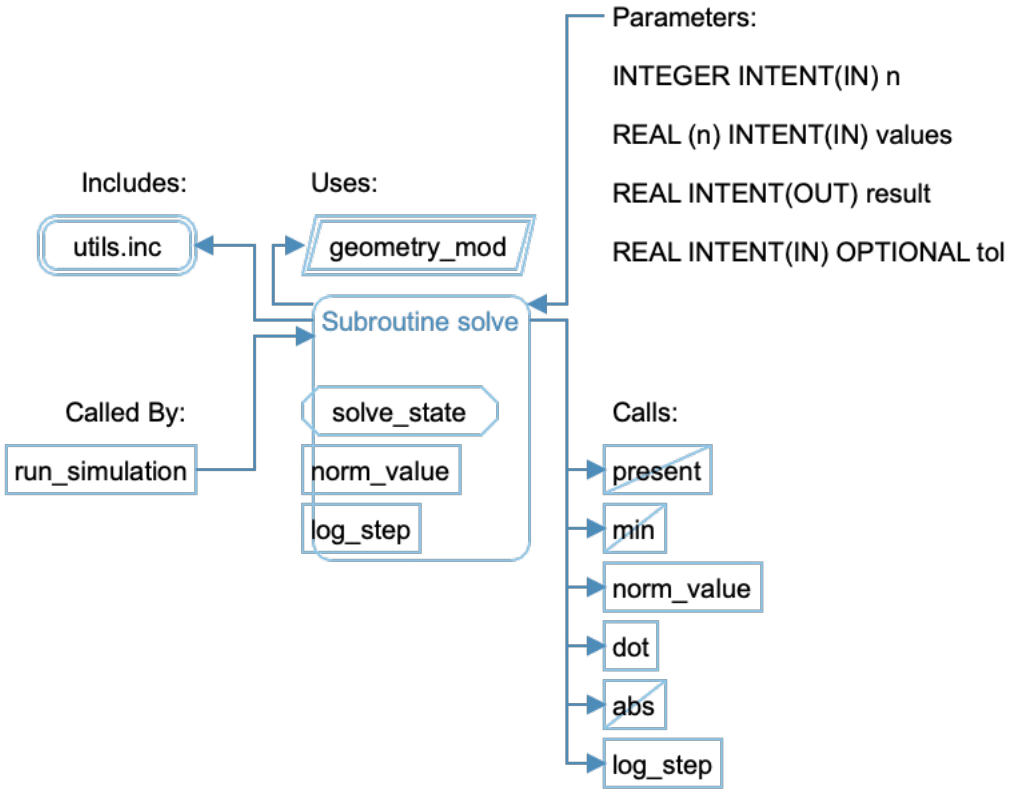
The main box lists what's declared directly inside the subprogram: internal common blocks, functions, and subroutines. Includes lists the files it includes; Uses lists the modules it uses; Parameters lists its formal parameters; Called By lists what calls it; Calls lists what it calls, including intrinsic functions.

Rooted at the subroutine solve below, the box shows the common block solve_state and the internal functions norm_value and log_step, all declared directly inside solve. Includes shows utils.inc; Uses shows geometry_mod; Parameters shows n, values, result, and tol; Called By shows run_simulation; Calls shows present, min, norm_value, dot, abs, and log_step.

See the following code and corresponding graph

```
integer, parameter :: MAX_ITER  = 500
real,    parameter :: SMALL_VAL = 1.0e-10

integer :: included_calls
logical :: included_ready
common /included_state/ included_calls, included_ready

subroutine solve(n, values, result, tol)
    use geometry_mod, only: dot, Vector2
     implicit none
     include 'utils.inc'

     integer,          intent(in)  :: n
     real,             intent(in)  :: values(n)
     real,             intent(out) :: result
     real, optional,   intent(in)  :: tol

     integer :: i
     real    :: tolerance, sum_val, w
     type(Vector2) :: v

     integer :: state_iter
     logical :: state_conv
     real    :: state_res
     common /solve_state/ state_iter, state_conv, state_res

     tolerance = SMALL_VAL
     if (present(tol)) tolerance = tol

     state_iter = 0
     state_conv = .false.
     result     = 0.0
     sum_val    = 0.0

     do i = 1, min(n, MAX_ITER)
```

```
36          sum_val    = sum_val + norm_value(values(i), 1.0)
37          state_iter = state_iter + 1
38      end do
39
40      v%x    = sum_val
41      v%y    = 1.0
42      w      = dot(v, v)
43      if (abs(w)  MAX_ITER) return
44      end subroutine log_step
45
46  end subroutine solve
```



Back to Declaration

Declaration - Function

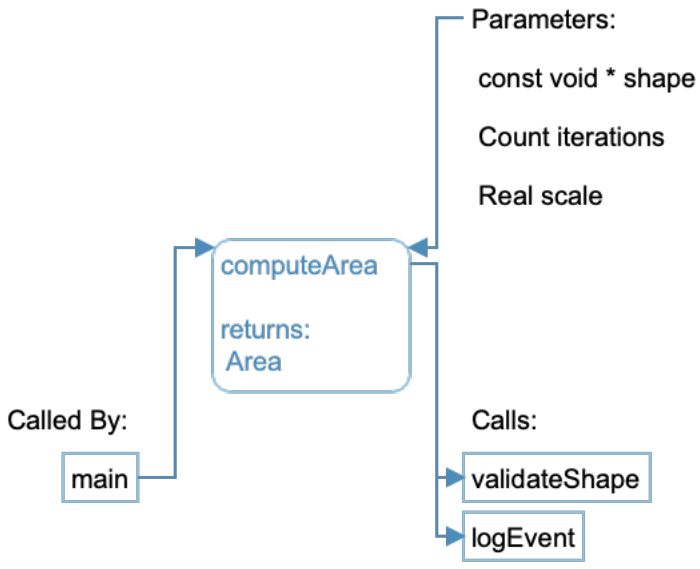
Show a diagram of a selected function's parameters, plus what calls it and what it calls.

The main box names the function and its return type. Parameters lists its formal parameters; Called By lists what calls it; Calls lists what it calls.

Rooted at computeArea below, Parameters shows const void * shape, Count iterations, and Real scale; Called By shows main, which calls computeArea; Calls shows validateShape and logEvent, which computeArea calls in turn.

See the following code and corresponding graph

```
1 // main.cpp
2 Area ca = computeArea(&c, 10, Geometry::scale_factor);
3
4 // shapes.cpp
5 static void logEvent(const char* msg, int level) {
6     (void)msg; (void)level;
7     ++cache_hits;
8 }
9
10 static void validateShape(const Shape* s) {
11     (void)s;
12 }
13
14 Area computeArea(const void* shape, Count iterations, Real scale) {
15     (void)iterations;
16     Area total = 0.0;
17     const Shape* s = static_cast(shape);
18     validateShape(s);
19     if (s) {
20         total = s->area() * scale;
21         logEvent("computeArea", 2);
22     }
23     return total;
24 }
```



Back to Declaration

Declaration - Java Class

Show a diagram of a selected Java class or interface's own members, plus its

superclass, subclasses, and interfaces.

The main box lists private members at the top, protected members in the middle, and public members at the bottom, with methods and fields on the left and nested types on the right; a method with no body of its own (abstract, or declared but not implemented, as in an interface) is drawn with a dashed border. Enabling Extends and Extended By adds the type's immediate superclass and subclasses outside the main box; Implements and Implemented By add the interfaces it implements and, for an interface, the classes that implement it.

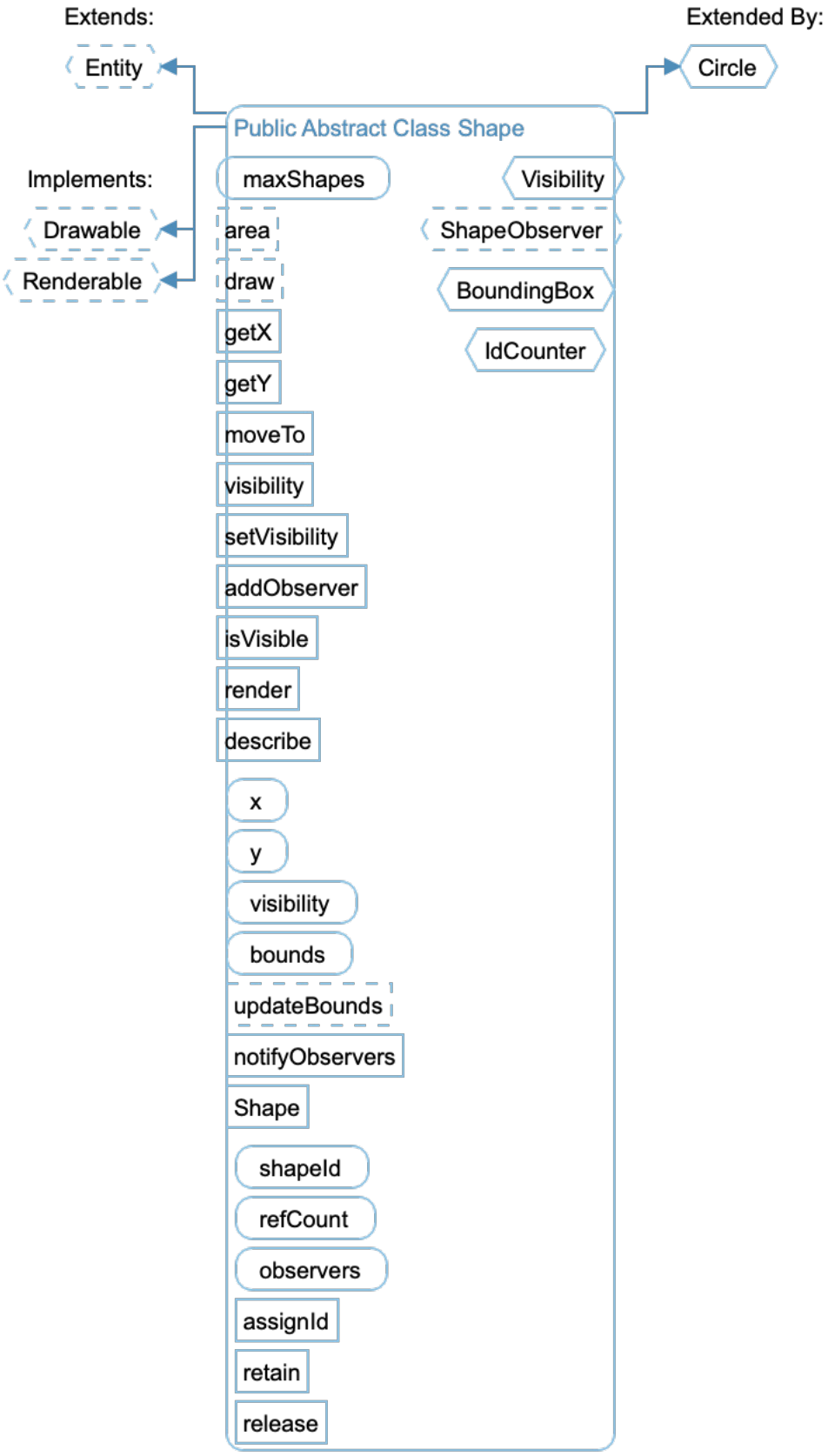
Rooted at the abstract class Shape below, the box shows the private fields shapeId, refCount, and observers and private methods assignId, retain, and release, and the private nested class IdCounter; the protected fields x, y, visibility, and bounds and protected methods updateBounds (dashed, abstract) and notifyObservers, and the protected nested class BoundingBox; and the public field maxShapes, the public methods area and draw (dashed, abstract), getX, getY, moveTo, visibility, setVisibility, addObserver, isVisible, render, and describe, and the public nested enum Visibility and interface ShapeObserver (dashed). Extends shows Entity; Extended By shows Circle; Implements shows Drawable and Renderable (both dashed, since they're interfaces).

See the following code and corresponding graph

```
1 public abstract class Entity {
2     private static int nextId = 0;
3
4     protected final int entityId;
5
6     protected Entity() {
7         this.entityId = ++nextId;
8     }
9
10     public    int    getId()      { return entityId; }
11     public abstract String describe();
12
13     static int allocateId() { return ++nextId; }
14 }
15
16 public interface Drawable {
17     void   draw();
18     double area();
19     default String drawDescription() { return "Drawable"; }
20 }
21
22 interface Renderable {
23     void    render(float alpha);
24     boolean isVisible();
25 }
26
27 public abstract class Shape
```

```
28      extends Entity implements Drawable, Renderable {
29
30      public enum Visibility { VISIBLE, HIDDEN, CLIPPED }
31
32      public interface ShapeObserver {
33          void onShapeChanged(Shape shape);
34      }
35
36      public static int maxShapes = 1000;
37
38      public abstract double area();
39      public abstract void   draw();
40
41      public double  getX()          { return x; }
42      public double  getY()          { return y; }
43      public void moveTo(double nx, double ny) {
44          x = nx; y = ny; updateBounds(); }
45      public Visibility visibility() { return visibility; }
46      public void    setVisibility(Visibility v) { visibility = v; }
47      public void addObserver(ShapeObserver obs) { observers.add(obs); }
48
49      public boolean isVisible()
50          { return visibility == Visibility.VISIBLE; }
51      public void    render(float alpha) { if (isVisible()) draw(); }
52
53      public String describe() {
54          return getClass().getSimpleName() + "(" + x + "," + y + ")"; }
55
56      protected static class BoundingBox {
57          public double minX, minY, maxX, maxY;
58          BoundingBox(double x, double y, double w, double h) {
59              minX = x; minY = y; maxX = x + w; maxY = y + h;
60          }
61      }
62
63      protected double     x, y;
64      protected Visibility visibility = Visibility.VISIBLE;
65      protected BoundingBox bounds;
66
67      protected abstract void  updateBounds();
68      protected void           notifyObservers() {
69          for (ShapeObserver o : observers) o.onShapeChanged(this);
70      }
71
72      private static class IdCounter {
73          private static int count = 0;
74          static int next() { return ++count; }
75      }
76
77      private int          shapeId;
78      private int          refCount   = 0;
79      private List observers = new ArrayList();
80
```

```
81      private void assignId()   { shapeId = IdCounter.next(); }
82      private void retain()     { refCount++; }
83      private void release()    { if (--refCount
```

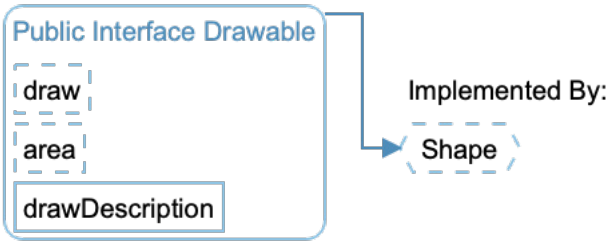


Rooted at the interface Drawable below, the box shows its methods draw and area (dashed, since interface methods have no body by default) and drawDescription

(not dashed, since it's a default method with its own implementation). Implemented By shows Shape (dashed, since Shape re-declares draw and area as abstract rather than implementing them itself).

See the following code and corresponding graph

```
1 public interface Drawable {
2     void   draw();
3     double area();
4     default String drawDescription() { return "Drawable"; }
5 }
6
7 interface Renderable {
8     void    render(float alpha);
9     boolean isVisible();
10 }
```



Back to Declaration

Declaration - Java File

Show a diagram of a selected Java file's own top-level type declarations, plus what it imports.

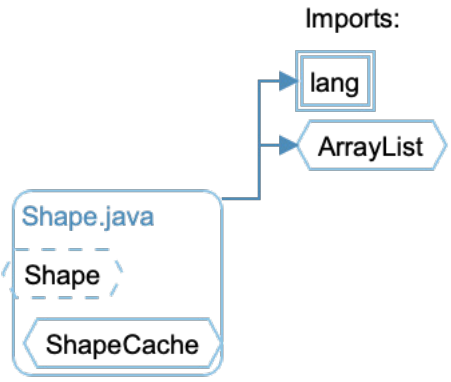
Enabling Public Members and Default Members lists the file's top-level classes and interfaces, split by accessibility: public types on the left, package-private (default) types on the right. Imports lists the packages and classes the file imports, including java.lang, which every Java file imports implicitly.

Rooted at Shape.java below, the box shows the public class Shape at the top and the default class ShapeCache at the bottom. Imports shows java.lang along with ArrayList, from the file's explicit import statements.

See the following code and corresponding graph

```
1 import java.util.List;
2 import java.util.ArrayList;
3
4 public abstract class Shape
5     extends Entity implements Drawable, Renderable {
6     /* ... */
```

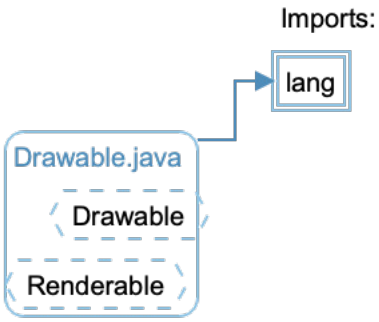
```
7 }
8
9 class ShapeCache {
10      /* ... */
11 }
```



Rooted at Drawable.java below, the box shows the public interface Drawable at the top and the default interface Renderable at the bottom. Imports shows only the implicit java.lang package, since the file has no explicit import statements.

See the following code and corresponding graph

```
1 public interface Drawable {
2     void   draw();
3     double area();
4     default String drawDescription() { return "Drawable"; }
5 }
6
7 interface Renderable {
8     void    render(float alpha);
9     boolean isVisible();
10 }
```



Back to Declaration

Declaration - Jovial File

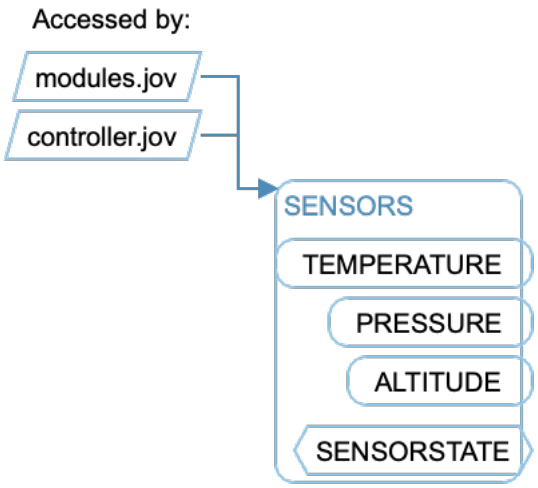
Show a diagram of a selected Jovial file or compool module's own declarations, plus what it accesses, is accessed by, and calls.

The main box lists the compool modules and external subroutines declared directly in the selected file or module on the left, and external variables and types on the right. Accessed by and Accesses list the files or modules that access, or are accessed by, a compool through a COMPOOL declaration; Calls lists the subroutines called.

Rooted at the compool module SENSORS below, the box shows its external variables TEMPERATURE, PRESSURE, and ALTITUDE and its external type SENSORSTATE. Accessed by shows controller.jov and modules.jov, both of which declare ! COMPOOL('SENSORS') to access it.

See the following code and corresponding graph

```
"Shared sensor data compool"
COMPOOL SENSORS;

ITEM TEMPERATURE U;
ITEM PRESSURE    U;
ITEM ALTITUDE    U;

TYPE SENSORSTATE STATUS(V(OFFLINE), V(READY), V(ACTIVE));

TERM
```



Rooted at the file sensors.jov below, the box shows the compool module SENSORS,

which the file declares.

See the following code and corresponding graph

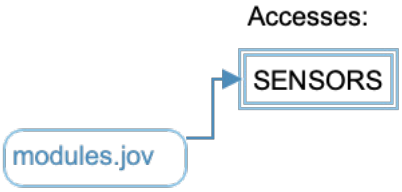
```
"Shared sensor data compool"
COMPOOL SENSORS;

ITEM TEMPERATURE U;
ITEM PRESSURE    U;
ITEM ALTITUDE    U;

TYPE SENSORSTATE STATUS(V(OFFLINE), V(READY), V(ACTIVE));

TERM
```



Rooted at the file modules.jov below, Accesses shows SENSORS, which the file's two programs each access via !COMPOOL('SENSORS'); neither ACQUIRE nor PROCESS appears as a module itself, since a Jovial module is a compool block, not a program.

See the following code and corresponding graph

```
START
  !COMPOOL('SENSORS');
  PROGRAM ACQUIRE;
  BEGIN
  "..."
  END
TERM

START
  !COMPOOL('SENSORS');
  PROGRAM PROCESS;
  BEGIN
  "..."
  END
TERM
```



Back to Declaration

Declaration - Pascal Class

Show a diagram of a selected Pascal class or interface's own members, plus its ancestor and descendant classes and interfaces.

The main box lists private members at the top, protected members in the middle, and public members at the bottom, with methods on the left and properties, published members, and nested types on the right. Enabling Extends and Extended By adds the class's immediate ancestor and descendants outside the main box; Implements and Implemented By add the interfaces it implements and, for an interface, the classes that implement it.

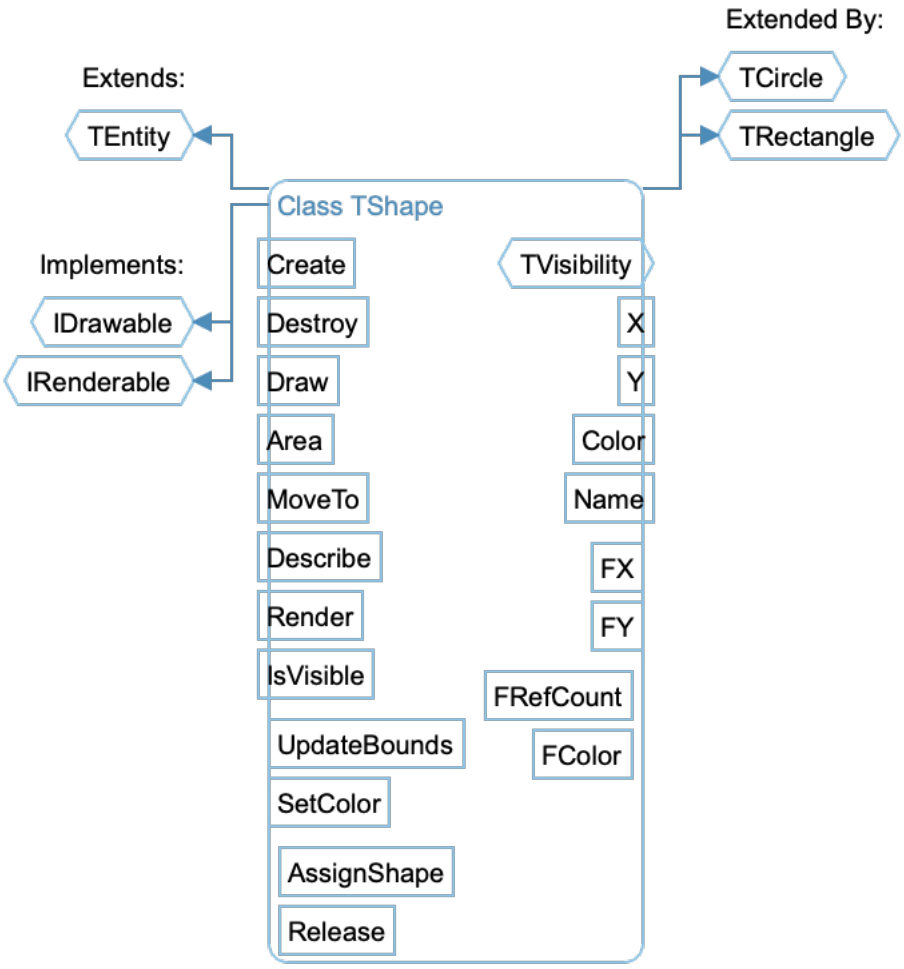
Rooted at the abstract class TShape below, the box shows the private methods AssignShape and Release and private data FRefCount and FColor; the protected methods UpdateBounds and SetColor and protected data FX and FY; and the public methods Create, Destroy, Draw, Area, MoveTo, Describe, Render, and IsVisible, the nested type TVisibility, and the properties X, Y, Color, and Name. Extends shows TEntity; Extended By shows TCircle and TRectangle; Implements shows IDrawable and IRenderable.

See the following code and corresponding graph

```
type
  IDrawable = interface
    procedure Draw;
    function  Area: Double;
  end;

  IRenderable = interface(IDrawable)
    procedure Render(Alpha: Single);
    function  IsVisible: Boolean;
   end;

   TEntity = class
   private
     FId: Integer;
   protected
```

```
16      procedure Initialize; virtual;
17    public
18      constructor Create;
19      destructor  Destroy; override;
20      function    Id: Integer;
21      function    Describe: string; virtual; abstract;
22    end;
23
24    TShape = class(TEntity, IDrawable, IRenderable)
25    private
26      FRefCount: Integer;
27      FColor:    string;
28      procedure AssignShape;
29      procedure Release;
30    protected
31      FX, FY: Double;
32      procedure UpdateBounds; virtual;
33      procedure SetColor(const AValue: string);
34    public
35      type TVisibility = (vsVisible, vsHidden, vsClipped);
36    public
37      constructor Create(AX, AY: Double); virtual;
38      destructor  Destroy; override;
39      procedure   Draw; virtual; abstract;
40      function    Area: Double; virtual; abstract;
41      procedure   MoveTo(AX, AY: Double);
42      function    Describe: string; override;
43      procedure   Render(Alpha: Single); virtual;
44      function    IsVisible: Boolean; virtual;
45      property X:     Double read FX     write FX;
46      property Y:     Double read FY     write FY;
47      property Color: string read FColor write SetColor;
48    published
49      property Name: string read FColor;
50    end;
51
52    TCircle = class(TShape)
53    private
54      FRadius: Double;
55    public
56      constructor Create(AX, AY, ARadius: Double); reintroduce;
57      procedure   Draw; override;
58      function    Area: Double; override;
59      property Radius: Double read FRadius write FRadius;
60    end;
61
62    TRectangle = class(TShape)
63    private
64      FWidth, FHeight: Double;
65    public
66      constructor Create(AX, AY, AWidth, AHeight: Double); reintroduce;
67      procedure   Draw; override;
68      function    Area: Double; override;
```

```
69      property Width:  Double read FWidth  write FWidth;
70      property Height: Double read FHeight write FHeight;
71    end;
```



Back to Declaration

Declaration - Pascal CompUnit Unit

Show a diagram of a selected Pascal compilation unit's own declarations, plus what it inherits and calls.

The main box lists the functions and procedures declared directly in the unit on the left, and its constants, types, and variables on the right. Inherits and Inherited by list the units named in the unit's uses clause, or that in turn use it; Calls lists what the unit's routines call.

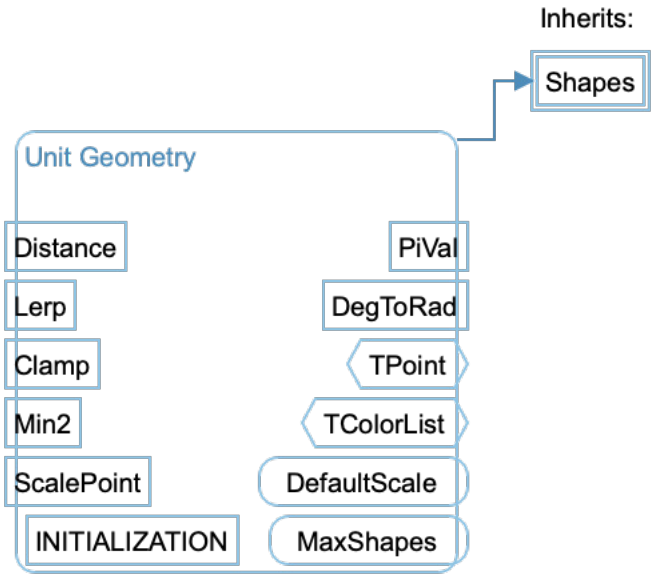
Rooted at the unit Geometry below, the box shows the functions Distance, Lerp, Clamp, and Min2, the function ScalePoint (which itself contains the local nested procedure NormalizeAxis), and the unit's initialization block; the constants PiVal and DegToRad, the types TPoint and TColorList, and the variables DefaultScale and

MaxShapes appear on the right. Clamp and Min2 are pulled in from common.inc through an $INCLUDE directive, so they count as part of Geometry itself. Inherits shows Shapes, from the unit's uses clause.

See the following code and corresponding graph

```
1 unit Geometry;
2
3 interface
4
5 uses
6   Shapes;
7
8 const
9   PiVal    = 3.14159265358979;
10   DegToRad = PiVal / 180.0;
11
12 type
13   TPoint = record
14     X, Y: Double;
15   end;
16
17   TColorList = array of string;
18
19 var
20   DefaultScale: Double;
21   MaxShapes:    Integer;
22
23 function Distance(const A, B: TPoint): Double;
24 function Lerp(A, B, T: Double): Double;
25 function Clamp(V, Lo, Hi: Double): Double;
26 function Min2(A, B: Double): Double;
27 function ScalePoint(const P: TPoint; Factor: Double): TPoint;
28
29 implementation
30
31 {$INCLUDE 'common.inc'}
32
33 function Distance(const A, B: TPoint): Double;
34 var
35   DX, DY: Double;
36 begin
37   DX := A.X - B.X;
38   DY := A.Y - B.Y;
39   Result := Sqrt(DX * DX + DY * DY);
40 end;
41
42 function Lerp(A, B, T: Double): Double;
43 begin
44   Result := A + (B - A) * Clamp(T, 0.0, 1.0);
45 end;
46
```

```
47 function ScalePoint(const P: TPoint; Factor: Double): TPoint;
48
49   procedure NormalizeAxis(var V: Double; Scale: Double);
50   begin
51     V := V * Scale;
52   end;
53
54 begin
55   Result := P;
56   NormalizeAxis(Result.X, Factor);
57   NormalizeAxis(Result.Y, Factor);
58 end;
59
60 initialization
61   DefaultScale := 1.0;
62   MaxShapes    := 256;
63
64 end.
```



Back to Declaration

Declaration - Pascal File Include

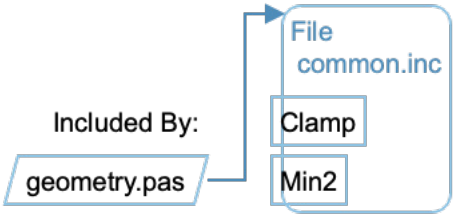
Show a diagram of a selected Pascal file's own declarations, plus what includes it.

The main box lists the functions, procedures, and compilation units defined directly in the file. Included By lists the files that include it through an $INCLUDE directive.

Rooted at common.inc below, the box shows the functions Clamp and Min2, which the file defines. Included By shows geometry.pas, which includes the file.

See the following code and corresponding graph

```
1 function Clamp(V, Lo, Hi: Double): Double;
2 begin
3   if V  Hi then Result := Hi
4   else Result := V;
5 end;
6
7 function Min2(A, B: Double): Double;
8 begin
9   if A
```



Back to Declaration

Declaration - Pascal Sql Table

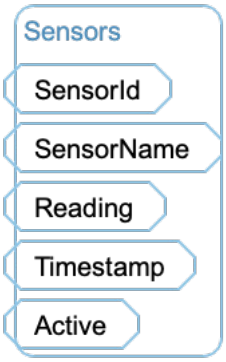
Show a diagram of a selected embedded SQL table's own columns.

The main box lists the table's columns.

Rooted at the table Sensors below, the box shows its columns SensorId, SensorName, Reading, Timestamp, and Active.

See the following code and corresponding graph

```
EXEC SQL CREATE TABLE Sensors (
   SensorId    INTEGER,
   SensorName  VARCHAR(64),
   Reading     FLOAT,
   Timestamp   INTEGER,
   Active      INTEGER
);
```



Back to Declaration

Declaration - Python Class

Show a diagram of a selected Python class's own methods and variables, plus its base and derived classes.

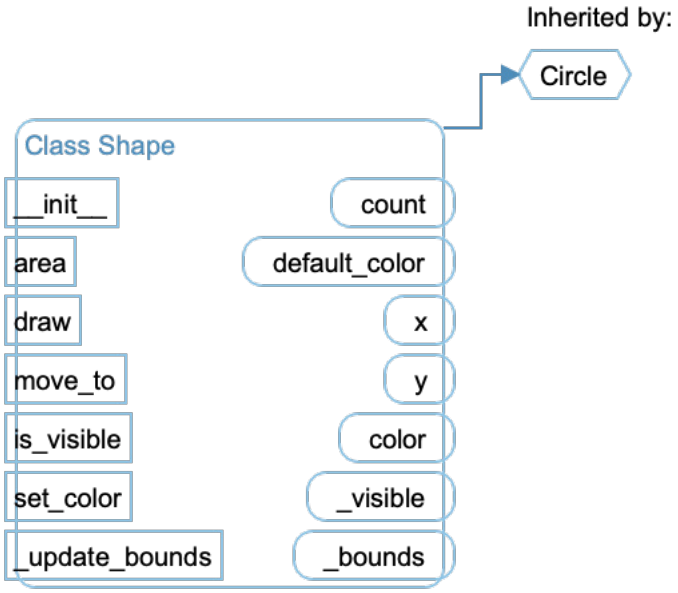
The main box lists the class's methods on the left and its class and instance variables on the right; Python has no visibility keywords, so a name starting with an underscore is treated the same as any other member. Enabling Extends and Extended By adds the class's immediate base classes and subclasses outside the main box.

Rooted at the class Shape below, the box shows its methods __init__, area, draw, move_to, is_visible, set_color, and _update_bounds, and its variables count, default_color, x, y, color, _visible, and _bounds. Shape has no explicit base class, so nothing appears under Extends; Extended By shows Circle.

See the following code and corresponding graph

```
1 class Shape:
2     count         = 0
3     default_color = Color.RED
4
5     def __init__(self, x, y, color=Color.RED):
6         self.x      = x
7         self.y      = y
8         self.color  = color
9         self._visible = True
10         self._bounds  = None
11         Shape.count  += 1
12
13     def area(self):
14         raise NotImplementedError
15
16     def draw(self):
17         raise NotImplementedError
18
19     def move_to(self, x, y):
```

```
20          self.x = x
21          self.y = y
22          self._update_bounds()
23
24      def is_visible(self):
25          return self._visible
26
27      def set_color(self, color):
28          self.color = color
29
30      def _update_bounds(self):
31          pass
32
33
34  class Circle(Shape):
35      def __init__(self, x, y, radius, color=Color.RED):
36          super().__init__(x, y, color)
37          self.radius = radius
38
39      def area(self):
40          return PI * self.radius ** 2
41
42      def draw(self):
43          pass
44
45      def scale(self, factor):
46          self.radius *= clamp(factor, 0.01, 100.0)
```



Back to Declaration

Declaration - Python File

Show a diagram of a selected Python file's own top-level declarations, plus what imports it and what it imports.

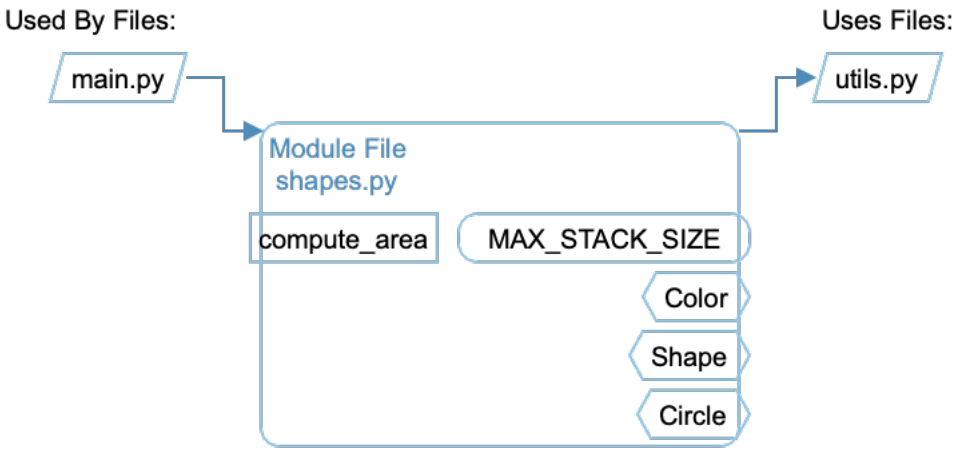
The main box lists the top-level functions defined directly in the file on the left, and its top-level classes and variables on the right. Used By Files and Uses Files list the files that import the file, or that it imports; Calls lists what the file's top-level code calls.

Rooted at shapes.py below, the box shows the function compute_area on the left, and the variable MAX_STACK_SIZE and the classes Color, Shape, and Circle on the right. Used By Files shows main.py, which imports shapes; Uses Files shows utils.py, which shapes.py imports from.

See the following code and corresponding graph

```
1 from utils import PI, clamp
2
3 MAX_STACK_SIZE = 100
4
5
6 def compute_area(shape):
7     ...
8
9
10 class Color:
11     RED   = 0
12     GREEN = 1
13     BLUE  = 2
14     ALPHA = 3
15
16
17 class Shape:
18     count         = 0
19     default_color = Color.RED
20
21     def __init__(self, x, y, color=Color.RED):
22         ...
23
24     def area(self):
25         ...
26
27     def draw(self):
28         ...
29
30     def move_to(self, x, y):
31         ...
32
33     def is_visible(self):
34         ...
35
36     def set_color(self, color):
```

```
37          ...
38
39      def _update_bounds(self):
40          ...
41
42
43  class Circle(Shape):
44      def __init__(self, x, y, radius, color=Color.RED):
45          ...
46
47      def area(self):
48          ...
49
50      def draw(self):
51          ...
52
53      def scale(self, factor):
54          ...
```



Back to Declaration

Declaration - Web Class

Show a diagram of a selected PHP or JavaScript class, or PHP interface's own members, plus its base classes, derived classes, and interfaces.

The main box lists private members at the top, protected members in the middle, and public members at the bottom, with methods on the left and properties and constants on the right; PHP has all three visibility levels, while JavaScript only distinguishes private (#-prefixed) from public. Enabling Base Classes and Derived Classes adds the type's immediate ancestors and descendants outside the main box; Implements and Implemented By add the PHP interfaces it implements and, for an interface, the classes that implement it.

Rooted at the abstract PHP class Shape below, the box shows the private methods

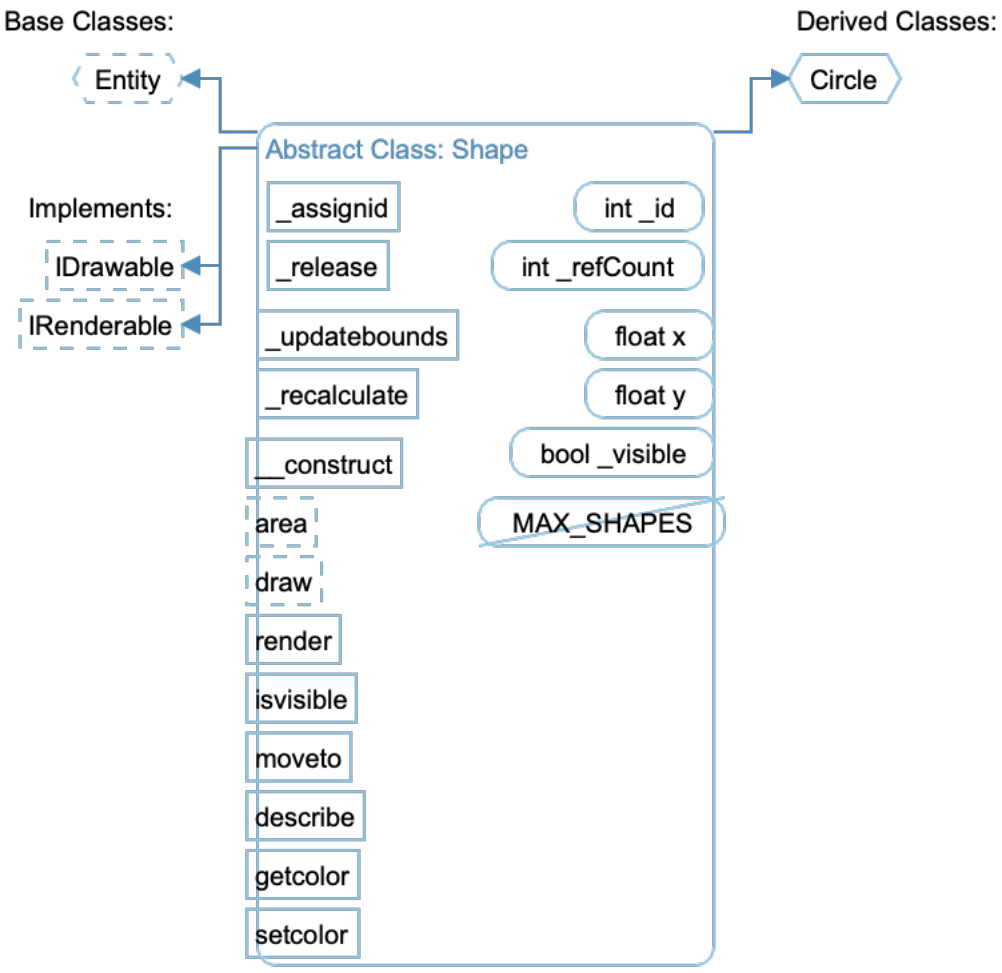
_assignId and _release and private properties $_id and $_refCount; the protected methods _updateBounds and _recalculate and protected properties $x, $y, and $_visible; and the public methods __construct, area and draw (both abstract), render, isVisible, moveTo, describe, getColor, and setColor, and the public constant MAX_SHAPES. Base Classes shows Entity (abstract); Derived Classes shows Circle; Implements shows IDrawable and IRenderable.

See the following code and corresponding graph

```
interface IBase {
    public function describe(): string;
}

interface IDrawable extends IBase {
    public function draw(): void;
    public function area(): float;
}

interface IRenderable {
    public function render(float $alpha): void;
    public function isVisible(): bool;
}

abstract class Entity {
    private static int $nextId = 0;
    protected int $entityId;

    public function __construct() {
        $this->entityId = ++self::$nextId;
    }
}

abstract class Shape extends Entity
    implements IDrawable, IRenderable {
    const MAX_SHAPES = 256;

    private int $_id;
    private int $_refCount;

    protected float $x;
    protected float $y;
    protected bool $_visible;

    private function _assignId(): void {
        $this->_id = rand(1, 9999);
```

```
    }

    private function _release(): void {
        $this->_refCount--;
    }

    protected function _updateBounds(): void {}

    protected function _recalculate(): void {}

    public function __construct(float $x, float $y) {
        parent::__construct();
        $this->x = $x;
        $this->y = $y;
        $this->_visible = true;
        $this->_refCount = 1;
        $this->_assignId();
    }

    abstract public function area(): float;
    abstract public function draw(): void;

    public function render(float $alpha): void {}

    public function isVisible(): bool {
        return $this->_visible;
    }

    public function moveTo(float $x, float $y): void {
        $this->x = clamp($x, -1000.0, 1000.0);
        $this->y = clamp($y, -1000.0, 1000.0);
        $this->_updateBounds();
    }

    public function describe(): string {
        return "Shape({$this->x}, {$this->y})";
    }

    public function getColor(): string { return "black"; }
    public function setColor(string $color): void {}
}

class Circle extends Shape {
    private float $radius;
```

```
    public function __construct(float $x, float $y, float $radius) {
        parent::__construct($x, $y);
        $this->radius = clamp($radius, 0.01, 1000.0);
    }

    public function area(): float {
        return PI * $this->radius ** 2;
    }

    public function draw(): void {}

    public function scale(float $factor): void {
        $this->radius *= clamp($factor, 0.01, 100.0);
    }
}
```



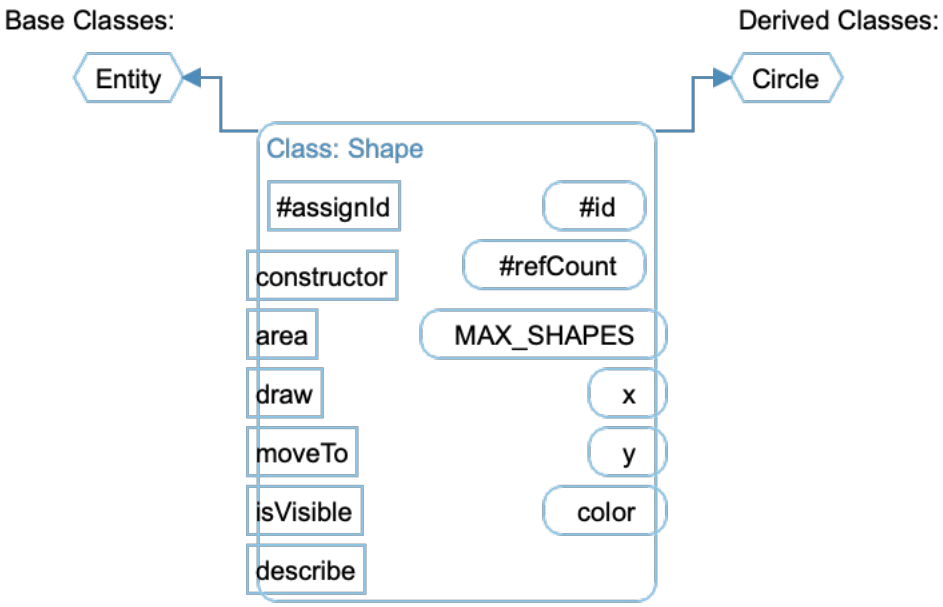
Rooted at the JavaScript class Shape below, the box shows the private method #assignId and private properties #id and #refCount; and the public methods

constructor, area, draw, moveTo, isVisible, and describe, and the public properties MAX_SHAPES, x, y, and color. JavaScript has no protected members, so the middle of the box is empty here. Base Classes shows Entity; Derived Classes shows Circle.

See the following code and corresponding graph

```
class Entity {
    static #nextId = 0;
    #entityId;

    constructor() {
        this.#entityId = ++Entity.#nextId;
    }

    describe() { return `Entity(${this.#entityId})`; }
}

class Shape extends Entity {
    static MAX_SHAPES = 256;

    #id;
    #refCount;

    x = 0;
    y = 0;
    color = 0;

    constructor(x, y, color = 0) {
        super();
        this.x = x;
        this.y = y;
        this.color = color;
        this.#refCount = 1;
        this.#assignId();
    }

    #assignId() {
        this.#id = Math.floor(Math.random() * 9999);
    }

    area() { return 0; }
    draw() {}

    moveTo(x, y) {
        this.x = clamp(x, -1000, 1000);
        this.y = clamp(y, -1000, 1000);
    }

    isVisible() { return true; }

    describe() { return `Shape(${this.x}, ${this.y})`; }
}
```

```
47
48 class Circle extends Shape {
49     #radius;
50
51     constructor(x, y, radius, color = 0) {
52         super(x, y, color);
53         this.#radius = clamp(radius, 0.01, 1000);
54     }
55
56     area() { return Math.PI * this.#radius ** 2; }
57     draw() {}
58
59     scale(factor) {
60         this.#radius *= clamp(factor, 0.01, 100);
61     }
62 }
```



Back to Declaration

Declaration - Web File

Show a diagram of a selected PHP or JavaScript file's own top-level declarations, plus what requires or imports it, and what it requires or imports.

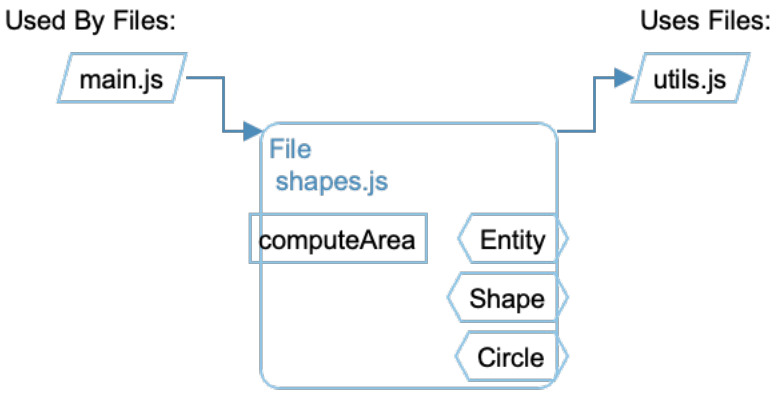
The main box lists the file's top-level functions on the left, and its top-level classes, constants, and variables on the right; PHP interfaces aren't included, only classes. Used By Files and Uses Files list the files that require, import, link, or use it, or that it in turn requires, imports, links, or uses; Calls lists what the file's top-level code calls.

Rooted at shapes.js below, the box shows the function computeArea on the left, and the classes Entity, Shape, and Circle on the right. Used By Files shows main.js,

which imports shapes.js; Uses Files shows utils.js, which shapes.js imports from.

See the following code and corresponding graph

```
1 import { clamp } from './utils.js';
2
3 export function computeArea(shape) {
4     return shape.area();
5 }
6
7 class Entity {
8     /* ... */
9 }
10
11 class Shape extends Entity {
12     /* ... */
13 }
14
15 class Circle extends Shape {
16     /* ... */
17 }
18
19 export { Entity, Shape, Circle };
```

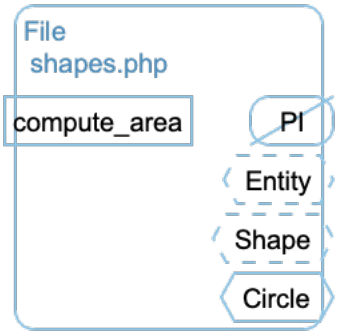


Rooted at shapes.php below, the box shows the function compute_area on the left, and the constant PI and the classes Entity and Shape (both dashed, since they're abstract) and Circle (solid, concrete) on the right; the interfaces IBase, IDrawable, and IRenderable declared in the same file don't appear, since this view only lists classes and constants, not interfaces. Used By Files shows app.php, which requires shapes.php; Uses Files shows utils.php, which shapes.php requires.

See the following code and corresponding graph

```
require_once 'utils.php';
```

```
const PI = 3.14159265358979;

function compute_area(Shape $shape): float {
    return $shape->area();
}

interface IBase {
    public function describe(): string;
}

interface IDrawable extends IBase {
    public function draw(): void;
    public function area(): float;
}

interface IRenderable {
    public function render(float $alpha): void;
    public function isVisible(): bool;
}

abstract class Entity {
    /* ... */
}

abstract class Shape extends Entity
    implements IDrawable, IRenderable {
    /* ... */
}

class Circle extends Shape {
    /* ... */
}
```



Back to Declaration

Declaration File

Languages: C#, C++, Fortran, Java, Jovial, Pascal, Python, Web
Targets: Classes, Functions, Objects, Subprograms, Types
Variants: Classic

Show the Declaration graph for the file a selected entity is defined in.

For a selected function, method, class, or similar entity, this jumps to the file it's defined in and shows that file's own Declaration graph.

For example, the Declaration File graph for logEvent is the Declaration graph for shapes.cpp, shown in Declaration - C Code File.

Declaration Tree

Languages: Ada, VHDL
Targets: Files, Packages, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the units and subprograms declared directly or indirectly inside a selected scope.

For a selected package, task, protected object, subprogram, or file (in Ada), or a selected architecture, entity, configuration, package, or subprogram (in VHDL), this shows an edge to each unit or subprogram declared immediately inside it, then recurses into each of those in turn, building a multi-level tree that mirrors the nesting of the source code.

Rooted at the Ada package Control below, the graph shows its nested declarations: the package Logging (which declares the procedure Write), the task Worker (which declares the entries Start and Stop), the protected object Mutex (which declares the entry Acquire, the procedure Release, and the function Is_Locked), and the procedure Run (which declares the nested function Double). The inverse graph, Declared In, shows the reverse relationship — the scope a selected unit is declared inside — but it's only available for Ada; there's no equivalent inverse graph for VHDL.

See the following code and corresponding graph

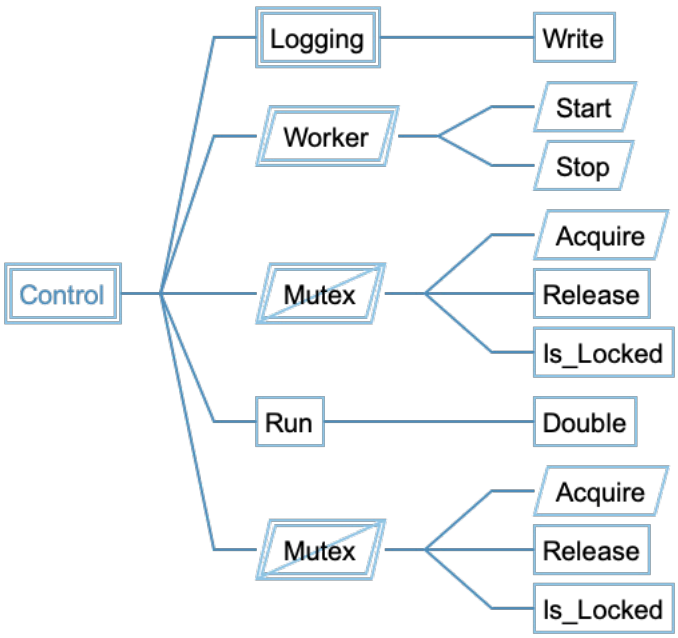
```
1 package Control is
2
3    package Logging is
4       procedure Write (Message : String);
5    end Logging;
```

```
6
7     task Worker is
8        entry Start (Id : Integer);
9        entry Stop;
10    end Worker;
11
12    protected Mutex is
13       entry Acquire;
14       procedure Release;
15       function Is_Locked return Boolean;
16    private
17       Locked : Boolean := False;
18    end Mutex;
19
20    procedure Run (Count : Integer);
21
22 end Control;
23
24 package body Control is
25
26    package body Logging is
27       procedure Write (Message : String) is
28          pragma Unreferenced (Message);
29       begin
30          null;
31       end Write;
32    end Logging;
33
34    task body Worker is
35    begin
36       loop
37          select
38             accept Start (Id : Integer) do
39                pragma Unreferenced (Id);
40             end Start;
41          or
42             accept Stop;
43             exit;
44          end select;
45       end loop;
46    end Worker;
47
48    protected body Mutex is
49       entry Acquire when not Locked is
50       begin
51          Locked := True;
52       end Acquire;
53
54       procedure Release is
55       begin
56          Locked := False;
57       end Release;
58
```

```
59       function Is_Locked return Boolean is
60       begin
61          return Locked;
62       end Is_Locked;
63    end Mutex;
64
65    procedure Run (Count : Integer) is
66       pragma Unreferenced (Count);
67
68       function Double (X : Integer) return Integer is
69       begin
70          return X * 2;
71       end Double;
72
73       Result : Integer := Double (0);
74       pragma Unreferenced (Result);
75    begin
76       null;
77    end Run;
78
79 end Control;
```

Declaration Tree - Classic

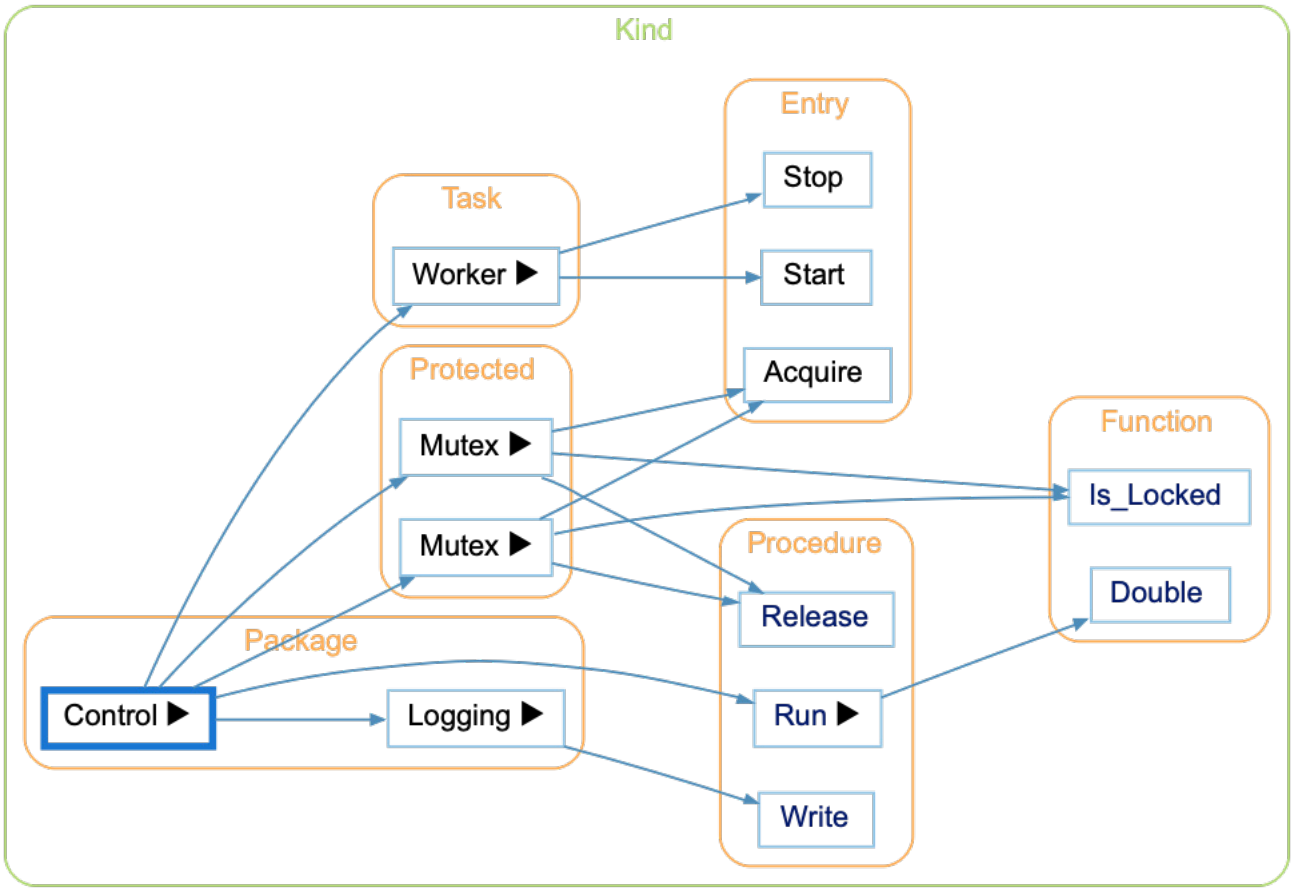
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Declaration Tree

Declaration Tree - Cluster

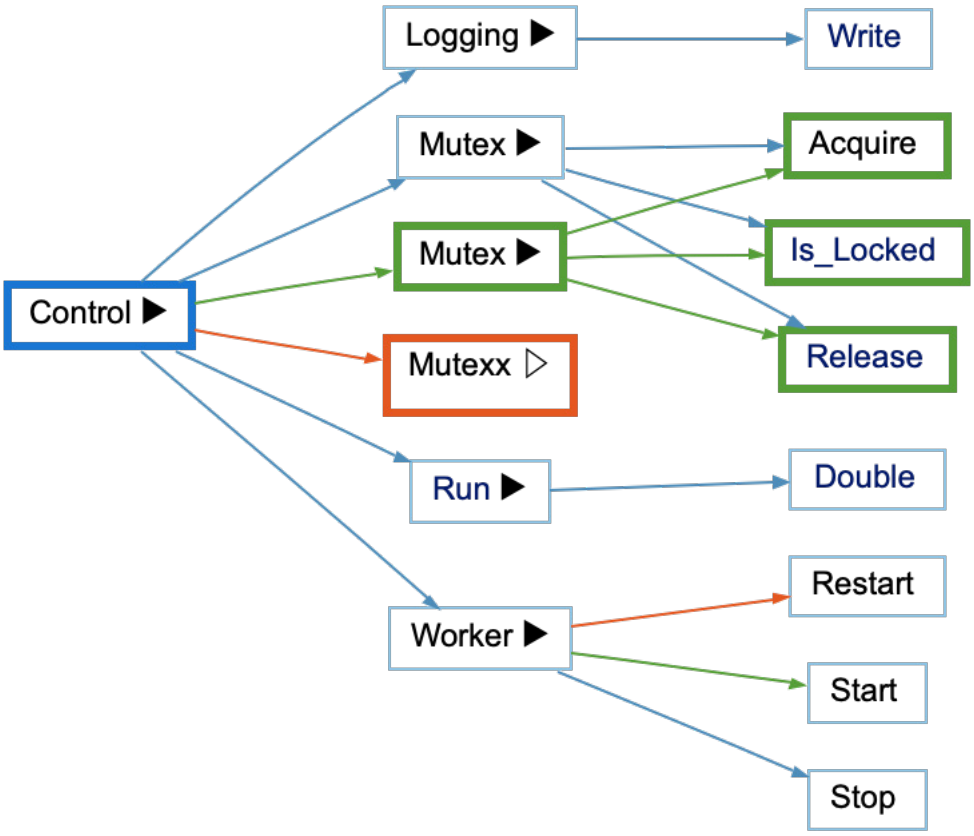
The Cluster variant groups entities by their containing class/file/architecture.



Back to Declaration Tree

Declaration Tree - Compare

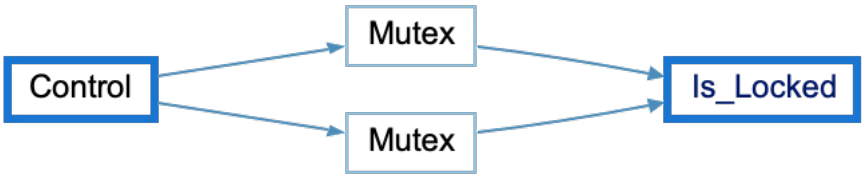
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Declaration Tree

Declaration Tree - Relationship

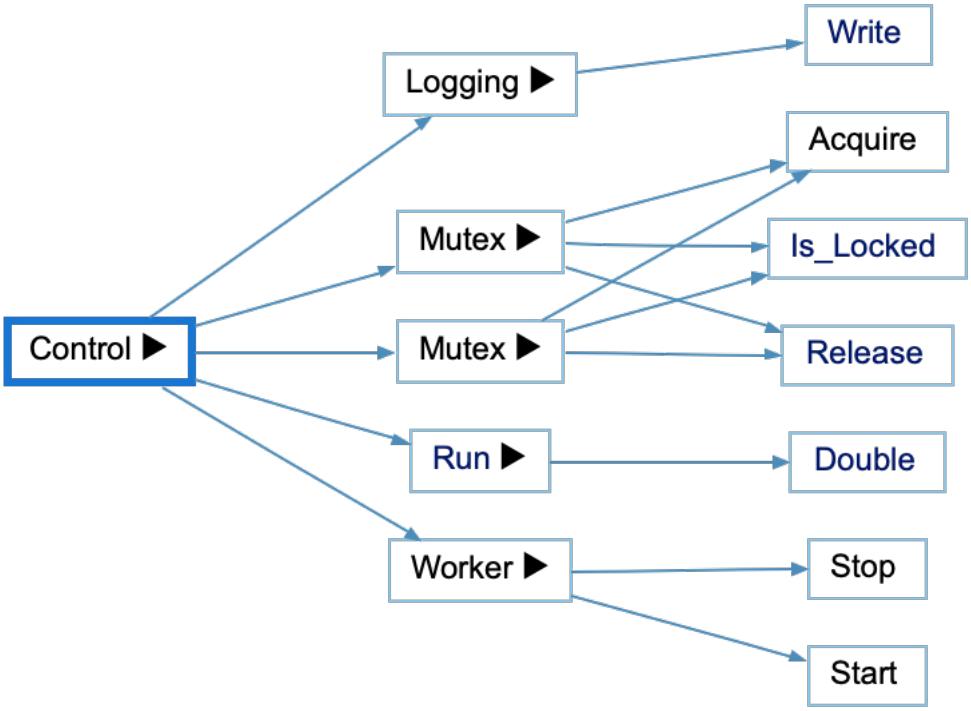
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Declaration Tree

Declaration Tree - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Declaration Tree

Declaration Tree & Declared In

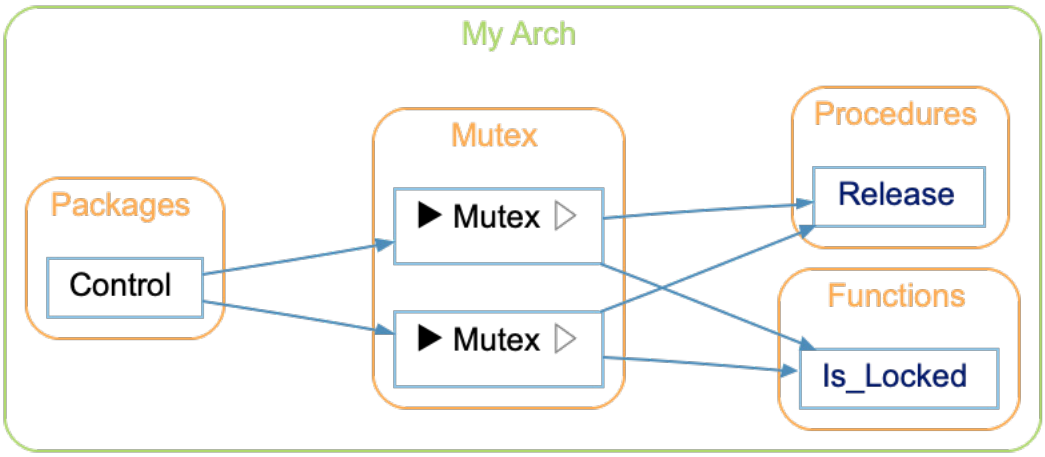
Languages: Ada
Targets: Architectures
Variants: Butterfly, Declaration Tree, Declared In, Internal

Show Declaration Tree and Declared In relationships for an architecture

See Declaration Tree and Declared In graphs for details

Declaration Tree & Declared In - Butterfly

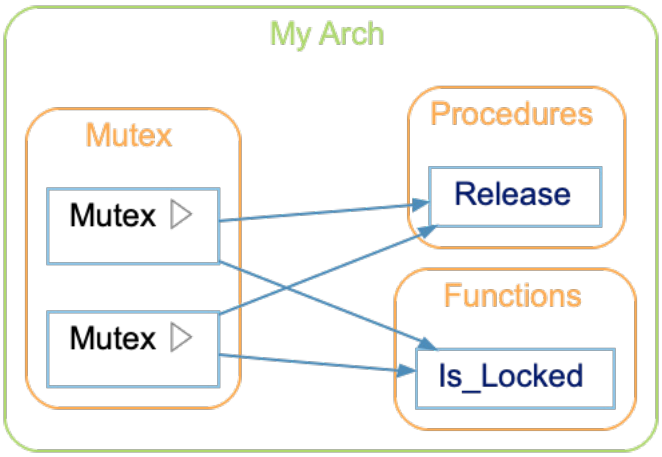
Show both Declaration Tree relationships from and Declared In relationships to entities in an architecture



Back to Declaration Tree & Declared In

Declaration Tree & Declared In - Declaration Tree

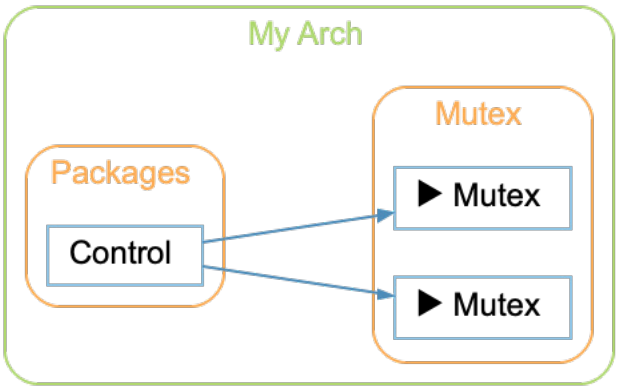
Show Declaration Tree relationships from entities in an architecture



Back to Declaration Tree & Declared In

Declaration Tree & Declared In - Declared In

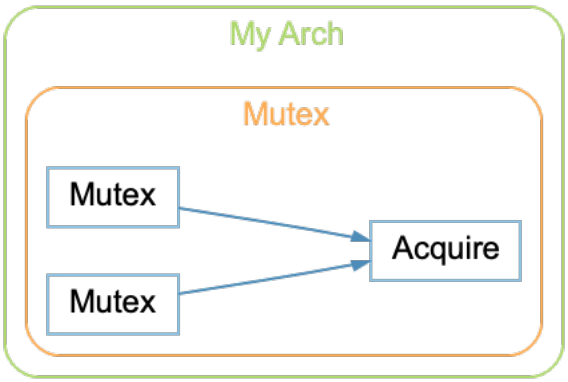
Show Declared In relationships to entities in an architecture



Back to Declaration Tree & Declared In

Declaration Tree & Declared In - Internal

Show Declaration Tree relationships between entities in an architecture



Back to Declaration Tree & Declared In

Declaration Type

Languages: C#, C++, Java, Jovial
Targets: Objects, Types
Variants: Classic

Show the Declaration graph for a selected variable's type, or a selected type alias's underlying type.

For a selected Java or Jovial variable, this jumps to its declared type and shows that type's own Declaration graph; for a selected C or C# type alias, this jumps to the type it aliases.

For example, the Declaration Type graph for ShapeAlias (declared as typedef Shape ShapeAlias; in shapes.h) is the Declaration graph for the class Shape, shown in Declaration - C Class.

Declared In

Languages: Ada
Targets: Packages, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the scope a selected unit is declared inside, and its own enclosing scope in turn.

For a selected task, protected object, package, subprogram, entry, or type, this shows an edge to the package, task, or protected object it's declared inside, then recurses upward into that scope's own enclosing scope, building a multi-level tree up to the top-level package.

Rooted at Is_Locked below, the graph shows an edge to Mutex, then continues up to Control. The inverse graph, Declaration Tree, shows the reverse relationship — everything declared inside the selected scope.

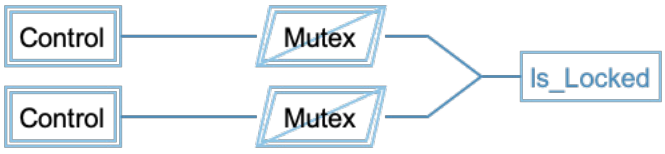
See the following code and corresponding graph

```
1 package Control is
2
3    protected Mutex is
4       entry Acquire;
5       procedure Release;
6       function Is_Locked return Boolean;
7    private
8       Locked : Boolean := False;
9    end Mutex;
10
11 end Control;
12
13 package body Control is
14
15    protected body Mutex is
16       entry Acquire when not Locked is
17       begin
18          Locked := True;
19       end Acquire;
20
21       procedure Release is
22       begin
23          Locked := False;
24       end Release;
```

```
25
26        function Is_Locked return Boolean is
27        begin
28           return Locked;
29        end Is_Locked;
30     end Mutex;
31
32 end Control;
```

Declared In - Classic

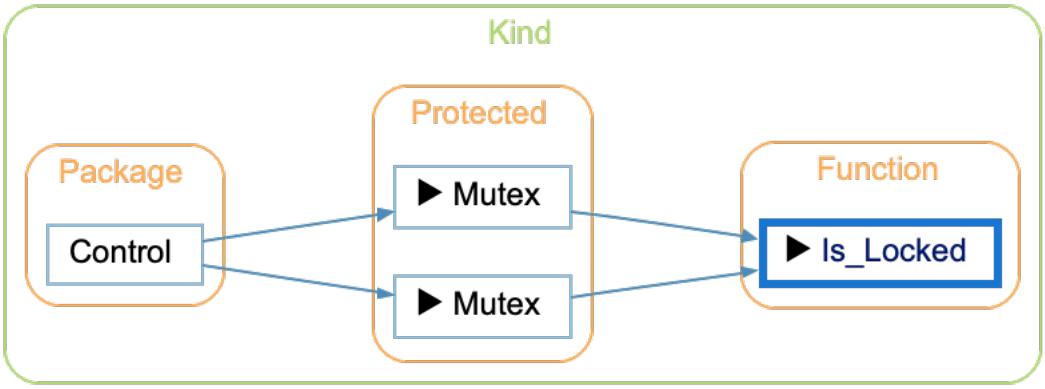
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Declared In

Declared In - Cluster

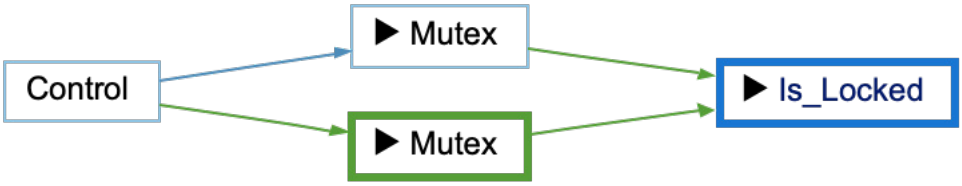
The Cluster variant groups entities by their containing class/file/architecture.



Back to Declared In

Declared In - Compare

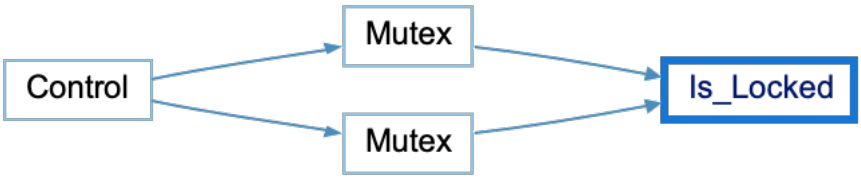
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Declared In

Declared In - Relationship

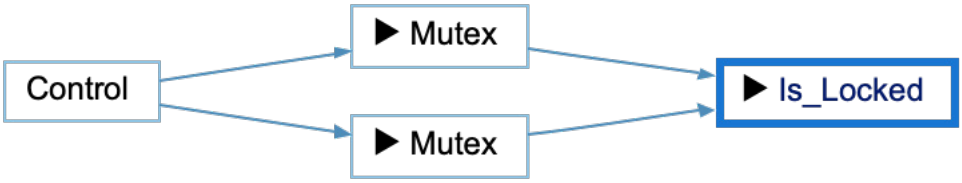
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Declared In

Declared In - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Declared In

Depended On By

Languages: Any
Targets: Classes, Files
Variants: Classic, Cluster, Compare

Show the files or classes that depend on the selected file or class.

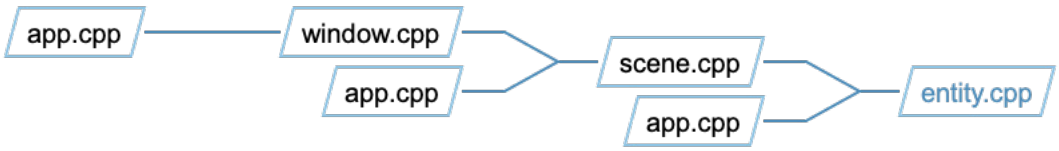
Dependencies map references to their owning file, class, or architecture. So a file's dependencies are the other files that own entities it references. For example, if a function defined in fileA calls a function defined in fileB, then fileA depends on fileB. References that contribute to dependencies vary by language, but generally include call, set, use, modify, and type references.

Dependencies always stay at the same grouping: file will only depend on other files, classes on other classes, and architectures on other architectures within the same root architecture.

Depended On By - Classic

Rooted at entity.cpp in the sample project, the graph shows two direct dependents — scene.cpp through 1 reference (a call to the update() function defined in entity.cpp), and app.cpp through 1 reference (a call to the Entity constructor defined in entity.cpp). In the next level, the two dependents of scene.cpp are shown — window.cpp through 2 references and app.cpp through 4 references. Finally, window.cpp is depended on by app.cpp through 5 references. See Depends On for the reverse direction.

The Classic variant shows the relationship as a simple tree (no clusters) using a custom layout algorithm instead of Graphviz.

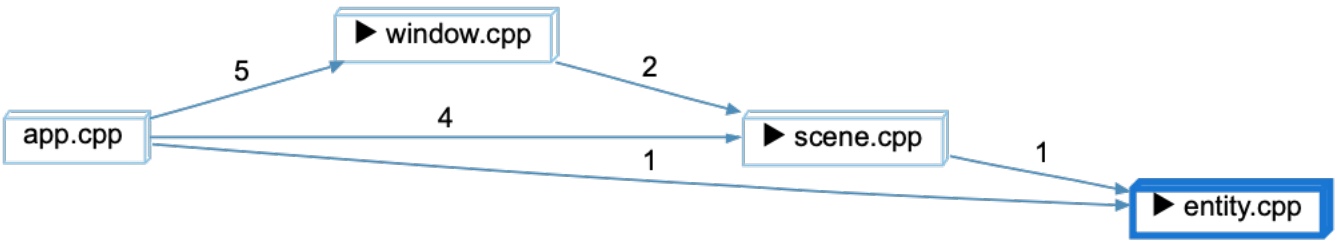


Back to Depended On By

Depended On By - Cluster

Rooted at entity.cpp in the sample project, the graph shows two direct dependents — scene.cpp through 1 reference (a call to the update() function defined in entity.cpp), and app.cpp through 1 reference (a call to the Entity constructor defined in entity.cpp). In the next level, the two dependents of scene.cpp are shown — window.cpp through 2 references and app.cpp through 4 references. Finally, window.cpp is depended on by app.cpp through 5 references. See Depends On for the reverse direction.

The Cluster variant shows each file/class once. In the GUI, the files/classes can be expanded to show the entities for each reference.

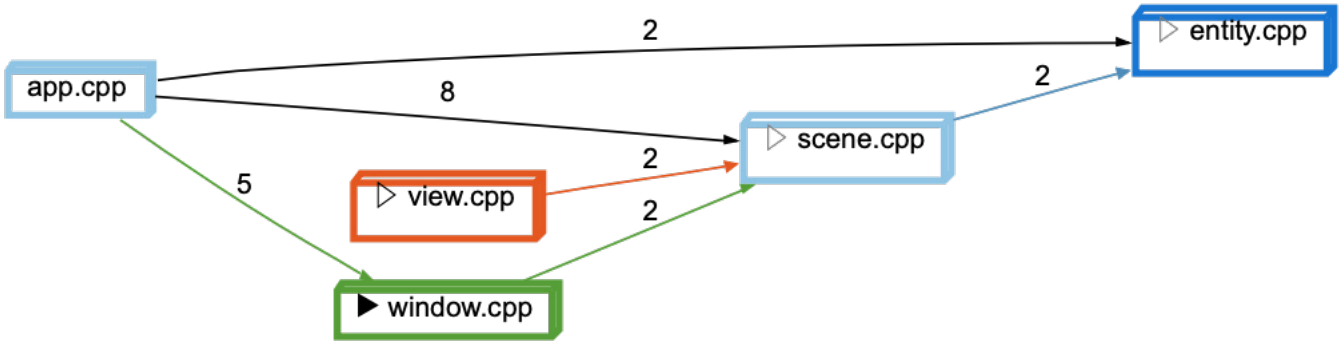


Back to Depended On By

Depended On By - Compare

Rooted at entity.cpp in the sample project, the graph shows two direct dependents — scene.cpp through 1 reference (a call to the update() function defined in entity.cpp), and app.cpp through 1 reference (a call to the Entity constructor defined in entity.cpp). In the next level, the two dependents of scene.cpp are shown — window.cpp through 2 references and app.cpp through 4 references. Finally, window.cpp is depended on by app.cpp through 5 references. See Depends On for the reverse direction.

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Depended On By

Dependencies

Languages: Any
Targets: Architectures
Variants: Butterfly, Depended On By, Depends On, Internal

Show dependencies for the selected architecture.

Dependencies map references to their owning file, class, or architecture. So a file's dependencies are the other files that own entities it references. For example, if a function defined in fileA calls a function defined in fileB, then fileA depends on fileB. References that contribute to dependencies vary by language, but generally include call, set, use, modify, and type references.

Dependencies always stay at the same grouping: file will only depend on other files, classes on other classes, and architectures on other architectures within the same root architecture.

See File Dependencies for an alternate view of architecture dependencies.

Dependencies - Butterfly

The Butterfly variant shows both directions, architectures the selected architecture depends on and is depended on by.

The sample project below is split into three architectures:

- lib: logging and vector-math utilities with no dependencies of its own
- api: an entity/scene/renderer layer built on lib
- ui: a small application built on api

See Graph Architecture for a graphical view of the architecture organization.

Rooted at api, the graph shows that api is depended on by ui through 22 references, and depends on lib through 27 references. In the GUI, architectures can be expanded to show the entities for each reference, and edges can be right clicked to directly view the references.



Back to Dependencies

Dependencies - Depended On By

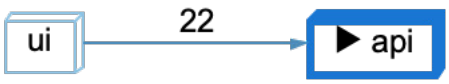
The Depended On By variant shows architectures the selected architecture is depended on by.

The sample project below is split into three architectures:

- lib: logging and vector-math utilities with no dependencies of its own
- api: an entity/scene/renderer layer built on lib
- ui: a small application built on api

See Graph Architecture for a graphical view of the architecture organization.

Rooted at api, the graph shows that api is depended on by ui through 22 references. In the GUI, architectures can be expanded to show the entities for each reference, and edges can be right clicked to directly view the references.



Back to Dependencies

Dependencies - Depends On

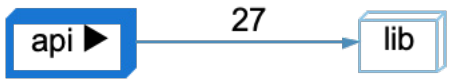
The Depends On variant shows architectures the selected architecture depends on.

The sample project below is split into three architectures:

- lib: logging and vector-math utilities with no dependencies of its own
- api: an entity/scene/renderer layer built on lib
- ui: a small application built on api

See Graph Architecture for a graphical view of the architecture organization.

Rooted at api, the graph shows that api depends on lib through 27 references. In the GUI, architectures can be expanded to show the entities for each reference, and edges can be right clicked to directly view the references.



Back to Dependencies

Dependencies - Internal

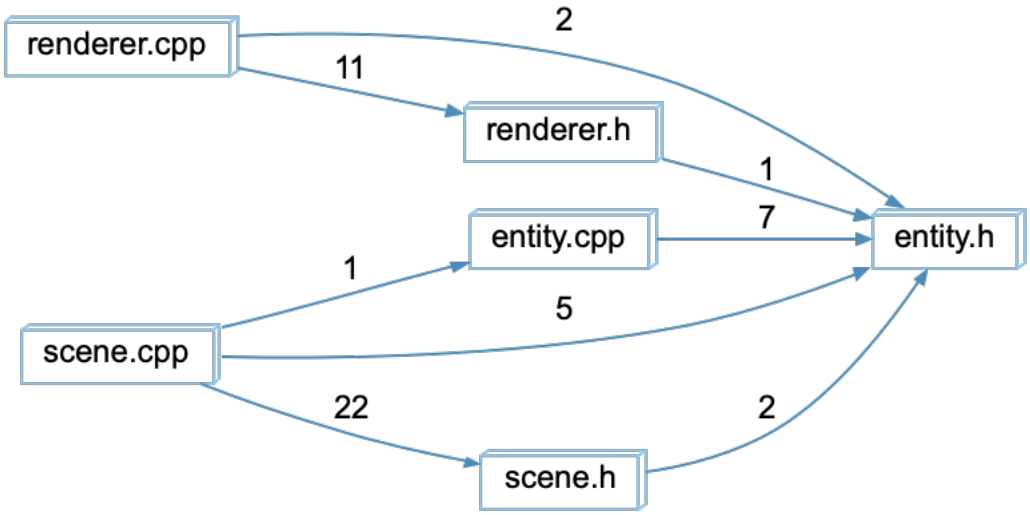
The Internal variant shows dependencies between the architecture's contents, both entities directly in the architecture and any nested architectures.

The sample project below is split into three architectures:

- lib: logging and vector-math utilities with no dependencies of its own
- api: an entity/scene/renderer layer built on lib
- ui: a small application built on api

See Graph Architecture for a graphical view of the architecture organization.

Rooted at api, the graph shows dependencies between the files in the folder. If api contained folders, those would be shown too. In the GUI, architectures can be expanded to show the entities for each reference, and edges can be right clicked to directly view the references.



Back to Dependencies

Depends On

Languages: Any
Targets: Classes, Files
Variants: Classic, Cluster, Compare

Show the files or classes the selected file or class depends on.

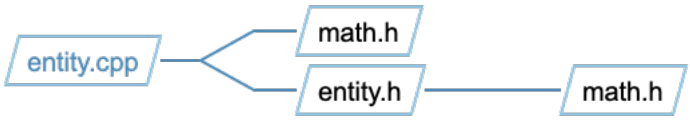
Dependencies map references to their owning file, class, or architecture. So a file's dependencies are the other files that own entities it references. For example, if a function defined in fileA calls a function defined in fileB, then fileA depends on fileB. References that contribute to dependencies vary by language, but generally include call, set, use, modify, and type references.

Dependencies always stay at the same grouping: file will only depend on other files, classes on other classes, and architectures on other architectures within the same root architecture.

Depends On - Classic

Rooted at entity.cpp in the sample project, the graph shows a direct dependency on entity.h (its own header) through 7 references and on math.h through 5 references. Plus, one step further, entity.h depends on math.h through 3 references. See Depended On By for the reverse direction.

The Classic variant shows the relationship as a simple tree (no clusters) using a custom layout algorithm instead of Graphviz.

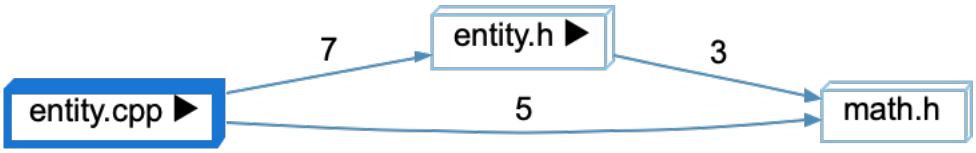


Back to Depends On

Depends On - Cluster

Rooted at entity.cpp in the sample project, the graph shows a direct dependency on entity.h (its own header) through 7 references and on math.h through 5 references. Plus, one step further, entity.h depends on math.h through 3 references. See Depended On By for the reverse direction.

The Cluster variant shows each file/class once. In the GUI, the files/classes can be expanded to show the entities for each reference.

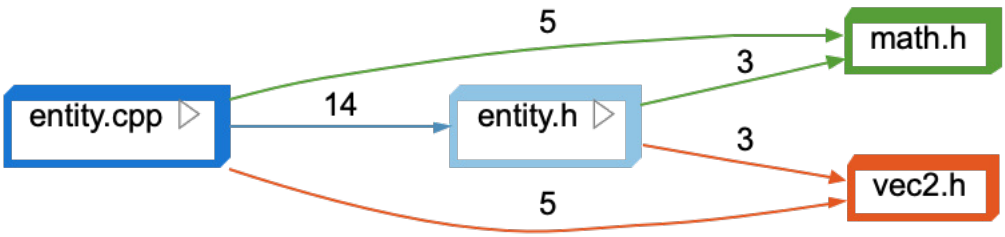


Back to Depends On

Depends On - Compare

Rooted at entity.cpp in the sample project, the graph shows a direct dependency on entity.h (its own header) through 7 references and on math.h through 5 references. Plus, one step further, entity.h depends on math.h through 3 references. See Depended On By for the reverse direction.

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Depends On

Derived Classes

Languages: C#, C++, Python, Web

Targets: Classes
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the classes that directly or indirectly inherit from a selected class.

For a selected class, struct, or interface, this shows an edge to each class that directly inherits from it, then recurses downward into each of those classes' own derived classes, building a multi-level tree of descendants.

Rooted at Vertebrate below, the graph shows edges to its two direct subclasses, Mammal and Bird, then continues into each of their own subclasses (Dog/Cat and Eagle/Hawk). The inverse graph, Base Classes, shows the reverse relationship — everything the selected class inherits from.

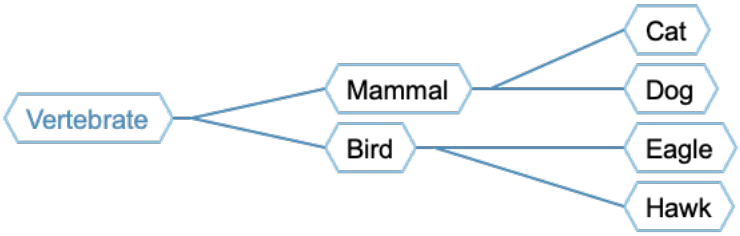
See the following code and corresponding graph

```
1 class Vertebrate {
2 public:
3     void describe() const;
4     void classify() const;
5 };
6
7 class Mammal : public Vertebrate {
8 public:
9     void describe() const override;
10 };
11
12 class Dog : public Mammal {
13 public:
14     void describe() const override;
15 };
16
17 class Cat : public Mammal {
18 public:
19     void describe() const override;
20 };
21
22 class Bird : public Vertebrate {
23 public:
24     void describe() const override;
25 };
26
27 class Eagle : public Bird {
28 public:
29     void describe() const override;
30 };
31
32 class Hawk : public Bird {
33 public:
34     void describe() const override;
```

```
35 };
```

Derived Classes - Classic

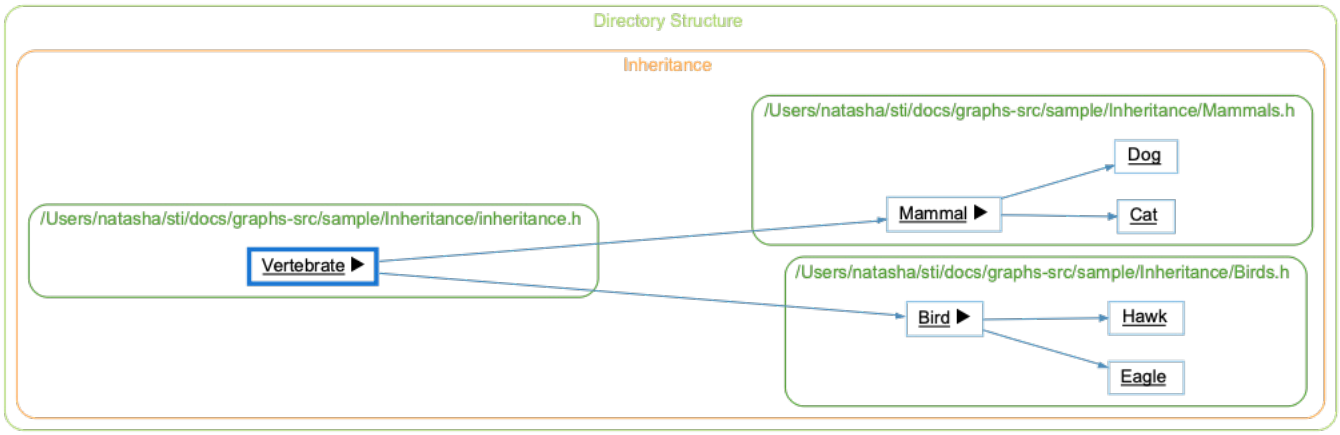
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Derived Classes

Derived Classes - Cluster

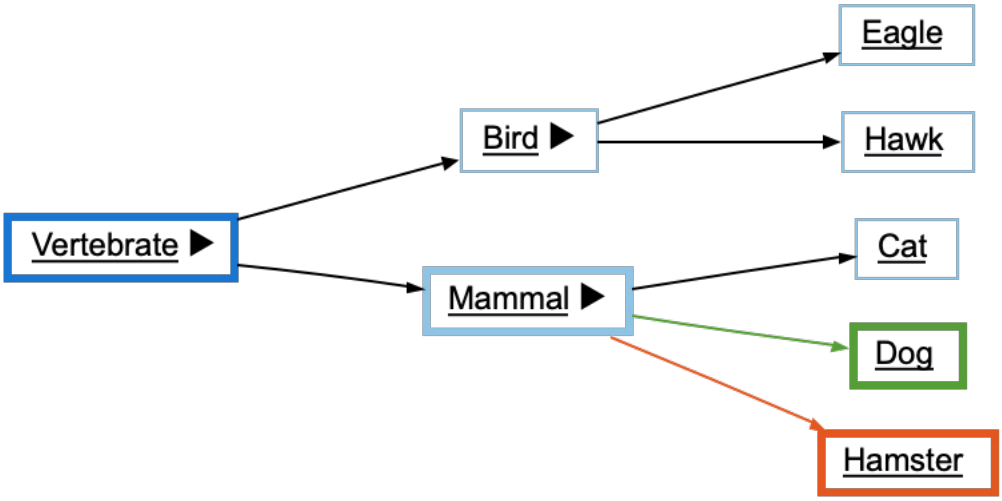
The Cluster variant groups entities by their containing class/file/architecture.



Back to Derived Classes

Derived Classes - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Derived Classes

Derived Classes - Relationship

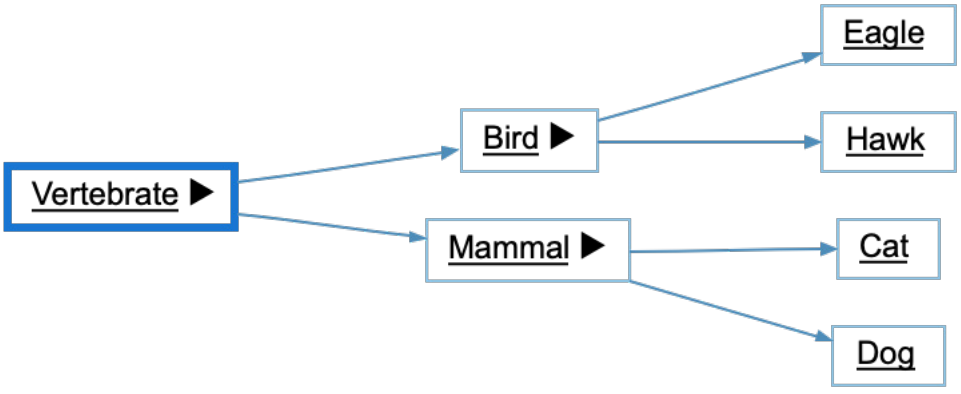
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Derived Classes

Derived Classes - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Derived Classes

Derived Classes & Base Classes

Languages: C#, C++, Python, Web

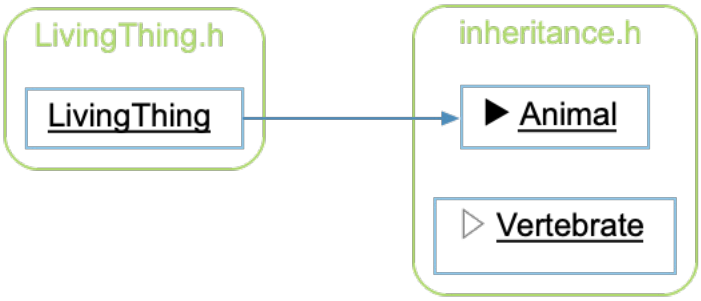
Targets: Architectures, Files
Variants: Base Classes, Butterfly, Derived Classes, Internal

Show Derived Classes and Base Classes relationships for a file or architecture

See Derived Classes and Base Classes graphs for details

Derived Classes & Base Classes - Base Classes

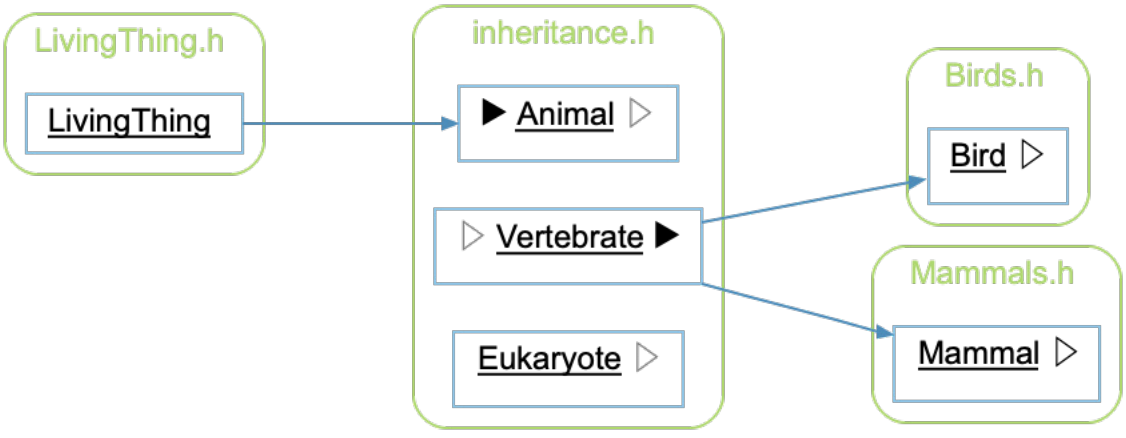
Show Base Classes relationships to entities in a file or architecture



Back to Derived Classes & Base Classes

Derived Classes & Base Classes - Butterfly

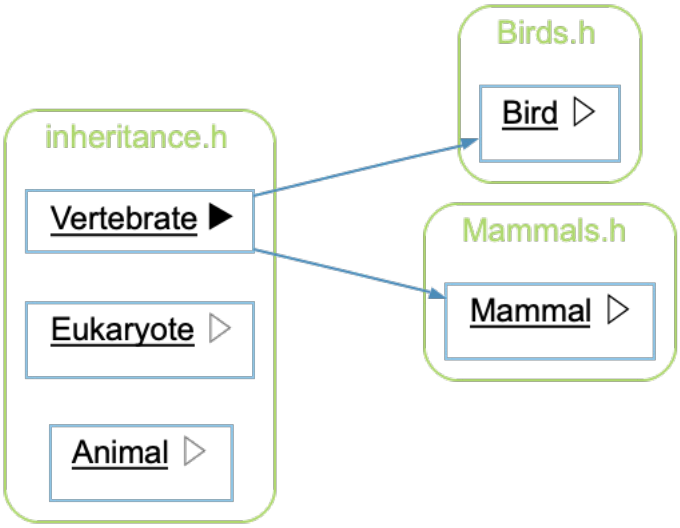
Show both Derived Classes relationships from and Base Classes relationships to entities in a file or architecture



Back to Derived Classes & Base Classes

Derived Classes & Base Classes - Derived Classes

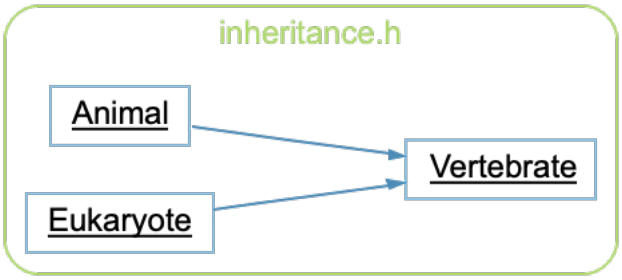
Show Derived Classes relationships from entities in a file or architecture



Back to Derived Classes & Base Classes

Derived Classes & Base Classes - Internal

Show Derived Classes relationships between entities in a file or architecture



Back to Derived Classes & Base Classes

Extended By

Languages: Fortran, Java, Pascal
Targets: Classes, Types
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the classes or interfaces that directly or indirectly extend a selected class or interface.

For a selected class, interface, or derived type, this shows an edge to each class that directly extends it, then recurses downward into whatever extends those classes in turn, building a multi-level tree of descendants.

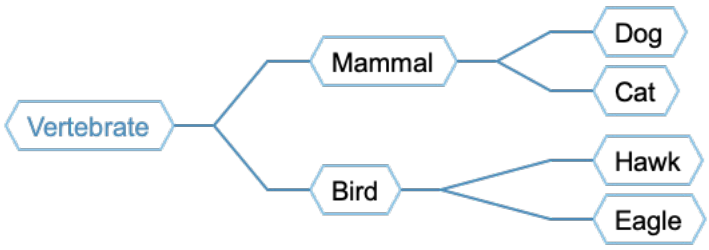
Rooted at the Java class Vertebrate below, the graph shows edges to Mammal and Bird, then continues into Dog/Cat and Eagle/Hawk. The inverse graph, Extends, shows the reverse relationship — everything the selected class extends.

See the following code and corresponding graph

```
1 public class Vertebrate {}
2
3 public class Mammal extends Vertebrate {}
4
5 public class Dog extends Mammal {}
6
7 public class Cat extends Mammal {}
8
9 public class Bird extends Vertebrate {}
10
11 public class Eagle extends Bird {}
12
13 public class Hawk extends Bird {}
```

Extended By - Classic

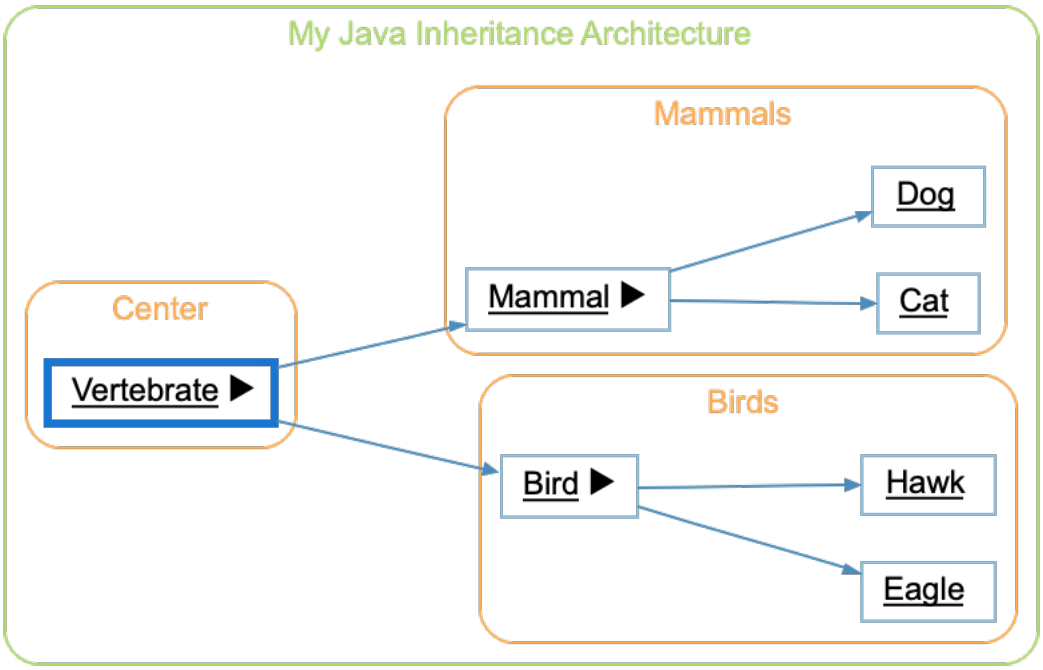
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Extended By

Extended By - Cluster

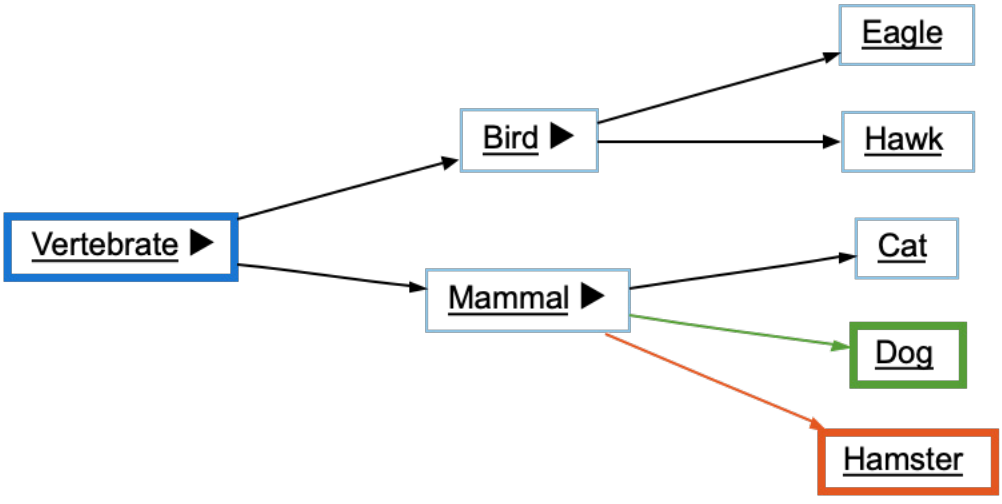
The Cluster variant groups entities by their containing class/file/architecture.



Back to Extended By

Extended By - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Extended By

Extended By - Relationship

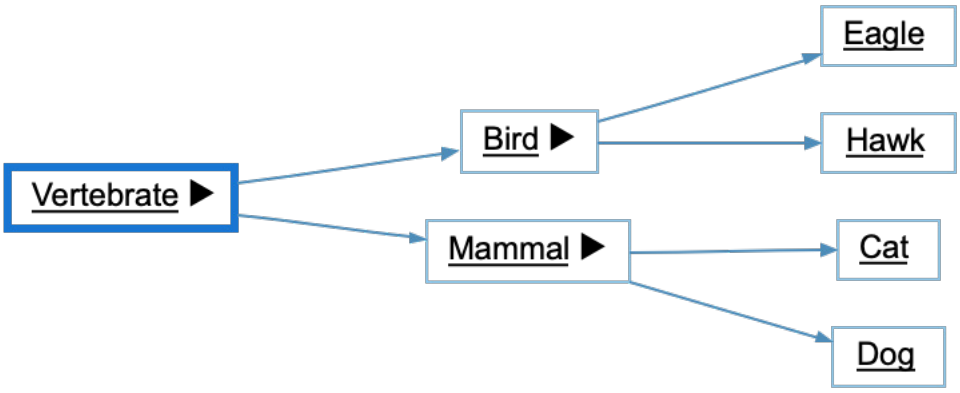
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Extended By

Extended By - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Extended By

Extended By & Extends

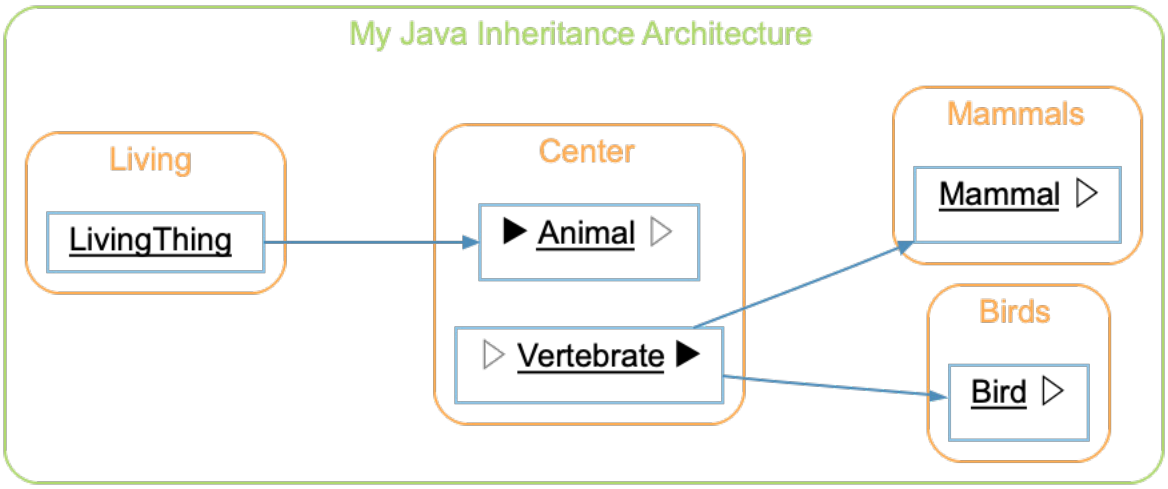
Languages: Fortran, Java, Pascal
Targets: Architectures, Files
Variants: Butterfly, Extended By, Extends, Internal

Show Extended By and Extends relationships for a file or architecture

See Extended By and Extends graphs for details

Extended By & Extends - Butterfly

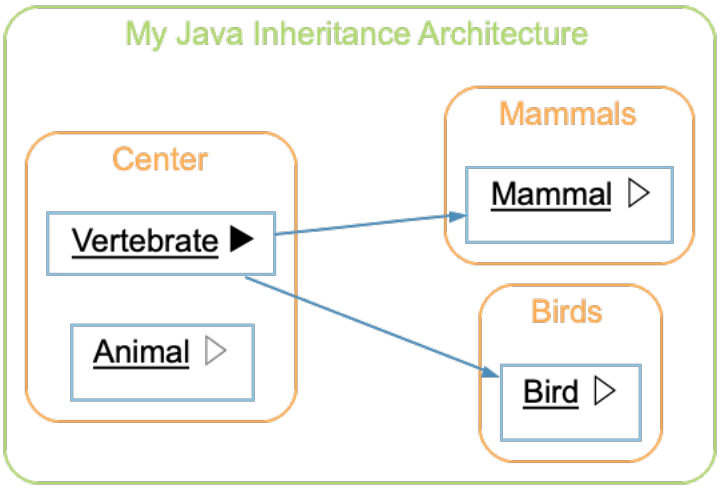
Show both Extended By relationships from and Extends relationships to entities in a file or architecture



Back to Extended By & Extends

Extended By & Extends - Extended By

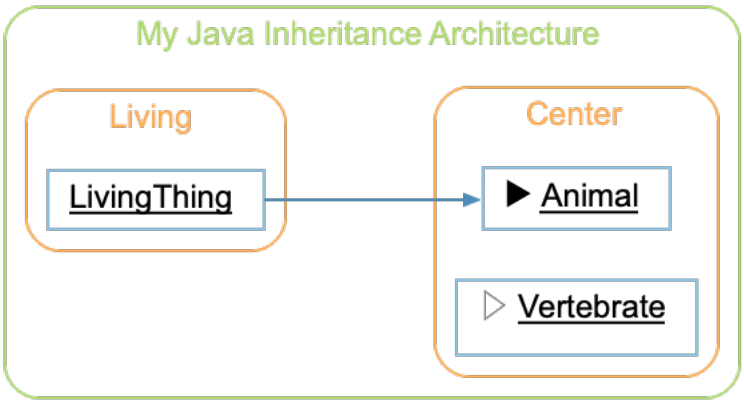
Show Extended By relationships from entities in a file or architecture



Back to Extended By & Extends

Extended By & Extends - Extends

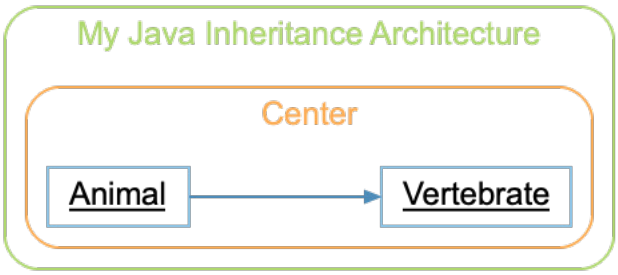
Show Extends relationships to entities in a file or architecture



Back to Extended By & Extends

Extended By & Extends - Internal

Show Extended By relationships between entities in a file or architecture



Back to Extended By & Extends

Extends

Languages: Fortran, Java, Pascal
Targets: Classes, Types
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the classes or interfaces a selected class or interface directly or indirectly extends.

For a selected class, interface, or derived type, this shows an edge to what it directly extends, then recurses upward into whatever that extends in turn, building a multi-level ancestry tree.

Rooted at the Java class Vertebrate below, the graph shows an edge to Animal, then continues up to LivingThing. The inverse graph, Extended By, shows the reverse

relationship — everything that extends the selected class.

See the following code and corresponding graph

```
1 public class LivingThing {}
2
3 public class Animal extends LivingThing {}
4
5 public class Vertebrate extends Animal {}
```

Extends - Classic

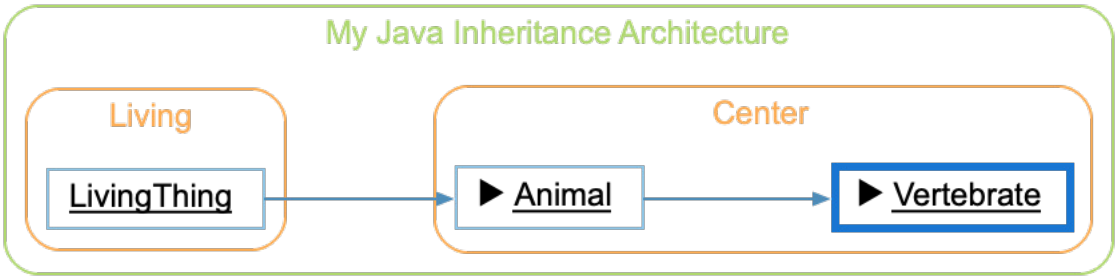
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Extends

Extends - Cluster

The Cluster variant groups entities by their containing class/file/architecture.



Back to Extends

Extends - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Extends

Extends - Relationship

The Relationship variant filters the graph to only paths connecting the two entities.



Back to Extends

Extends - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Extends

File Dependencies

Languages: Any
Targets: Architectures
Variants: Classic

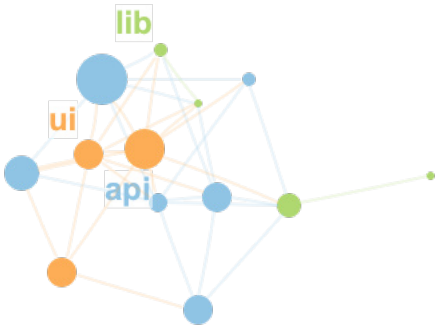
Show dependencies between files in the selected architecture.

See the Dependencies graph for a description of architecture dependencies. This graph shows the same relationships, but instead of clustering by child architectures, all files (recursively) are shown and the color indicates the child architecture. By default, the size of the file circle is scaled by the lines of code.

The sample project below is split into three architectures:

- lib (green): logging and vector-math utilities with no dependencies of its own
- api (blue): an entity/scene/renderer layer built on lib
- ui (orange): a small application built on api

See Graph Architecture for a graphical view of the architecture organization.



Graph Architecture

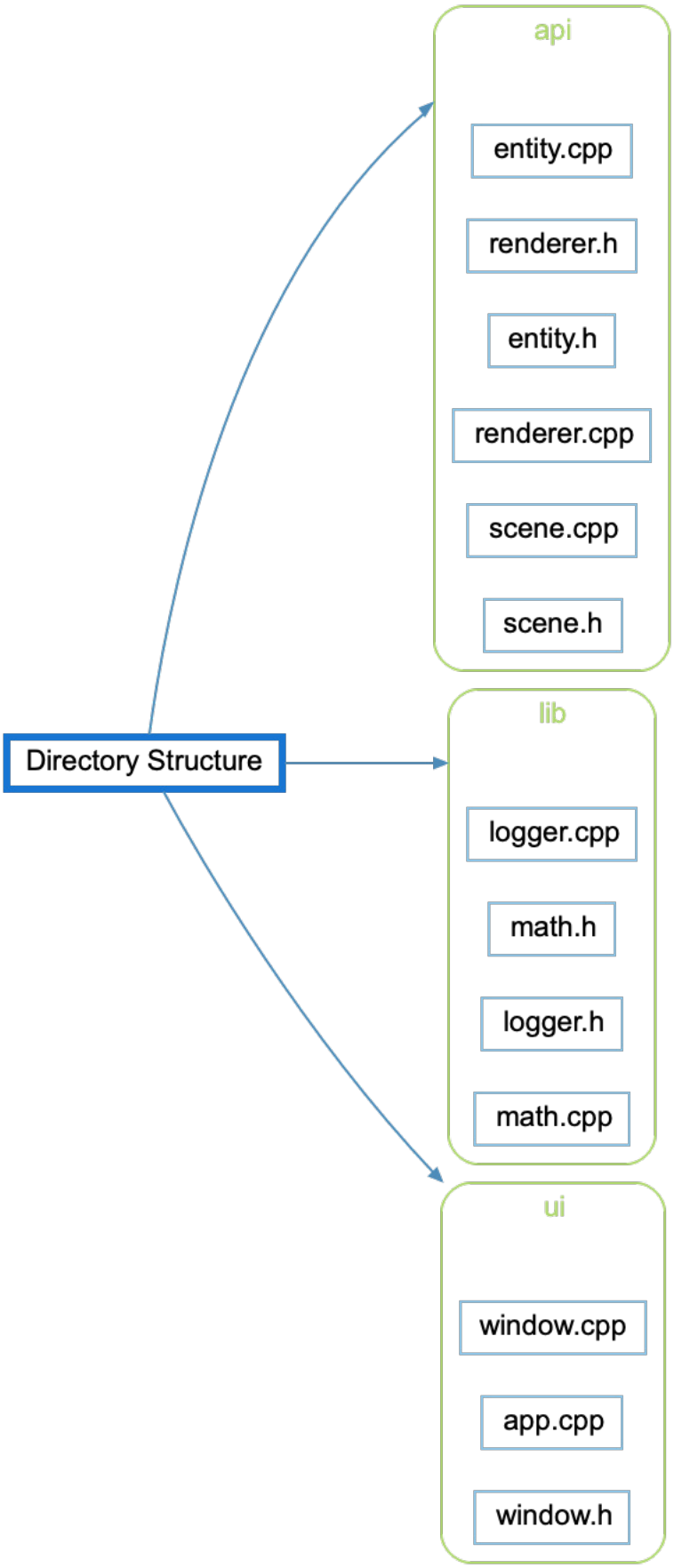
Languages: Any
Targets: Architectures
Variants: Classic

Show the selected architecture's hierarchy of child architectures and, optionally, the entities directly contained in each.

The root node is the selected architecture — a directory, or any other architecture defined for the project. Each child architecture is shown as its own node, connected by an edge to its parent, mirroring however the architecture is organized (by directory, by language construct, or by a custom grouping). Architecture nodes that contain entities are expandable. Expanding it shows the entities directly belonging to it, such as the files inside a folder.

See the Dependencies or File Dependencies graphs to explore how these same architectures depend on each other.

Rooted at Directory Structure below — the same sample project used by the Dependencies graphs — the graph shows the three top-level folders lib, api, and ui, each expanded to list its own files: lib contains logger.h, logger.cpp, math.h, and math.cpp; api contains entity.h, entity.cpp, renderer.h, renderer.cpp, scene.h, and scene.cpp; and ui contains window.h, window.cpp, and app.cpp.


api
entity.cpp
renderer.h
entity.h
renderer.cpp
scene.cpp
scene.h
lib
logger.cpp
math.h
logger.h
math.cpp
ui
window.cpp
app.cpp
window.h
Directory Structure

Imported By

Languages: Rust, Web
Targets: Files
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show what imports a selected JavaScript file or Rust entity, directly or indirectly.

For a selected JavaScript file, this shows an edge to each file that imports it, then recurses into each of those files' own importers, building a multi-level tree. For a Rust entity, this shows an edge to each entity that imports it through a use statement, then recurses into each of those entities' own importers the same way.

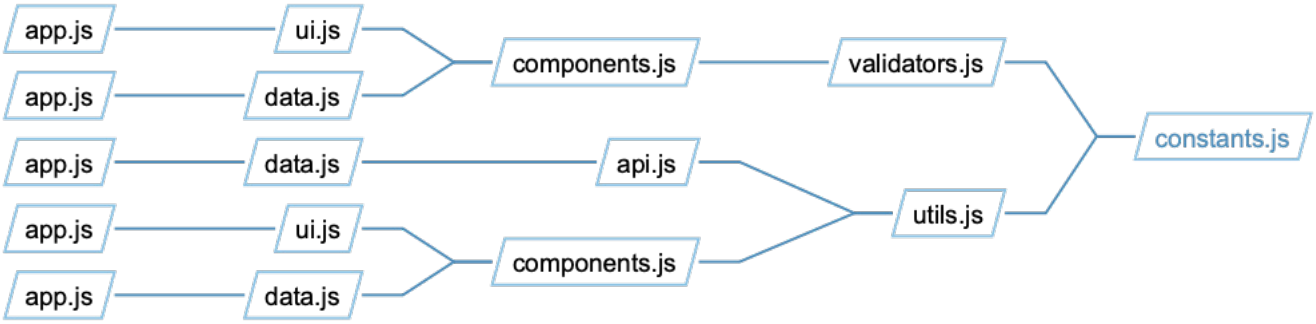
Rooted at constants.js below, the graph shows constants.js imported by utils.js and validators.js. utils.js is imported by api.js and components.js; validators.js is imported by components.js. components.js is imported by ui.js and data.js; api.js is imported by data.js. ui.js and data.js are each imported by app.js, which nothing imports. The inverse graph, Imports, shows the reverse relationship — everything the selected file or Rust entity imports, directly or indirectly.

See the following code and corresponding graph

```
1 // app.js
2 import { renderPage } from './ui.js';
3 import { loadData } from './data.js';
4
5 // ui.js
6 import { createButton, createForm } from './components.js';
7
8 // data.js
9 import { createForm } from './components.js';
10 import { fetchData } from './api.js';
11
12 // components.js
13 import { buildUrl } from './utils.js';
14 import { isValidInput } from './validators.js';
15
16 // api.js
17 import { buildUrl, delay } from './utils.js';
18
19 // utils.js
20 import { API_URL, TIMEOUT } from './constants.js';
21
22 // validators.js
23 import { MAX_RETRIES } from './constants.js';
24
25 // constants.js
```

Imported By - Classic

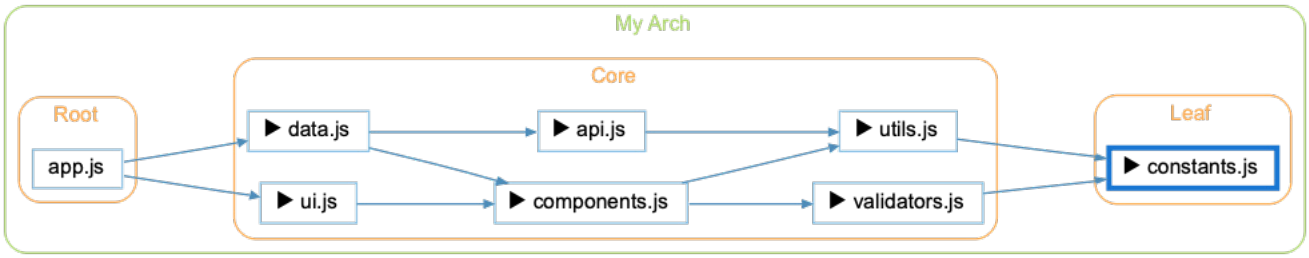
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Imported By

Imported By - Cluster

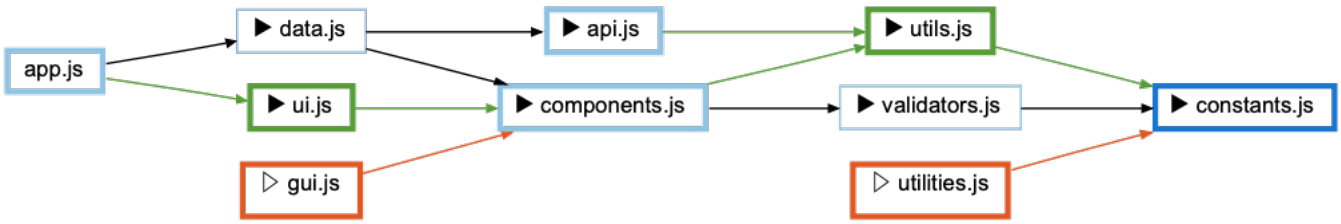
The Cluster variant groups entities by their containing class/file/architecture.



Back to Imported By

Imported By - Compare

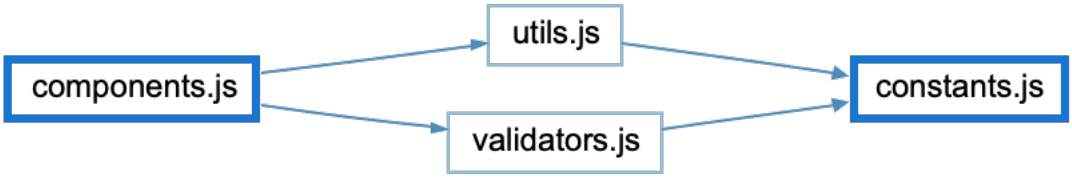
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Imported By

Imported By - Relationship

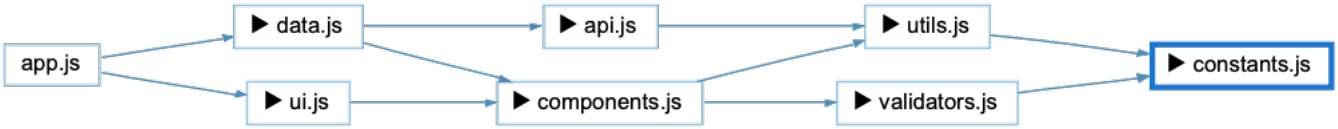
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Imported By

Imported By - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Imported By

Imports

Languages: Rust, Web
Targets: Files
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show what a selected JavaScript file or Rust entity imports, directly or indirectly.

For a selected JavaScript file, this shows an edge to each file it imports, then recurses into each of those files' own imports, building a multi-level tree. For a Rust entity, this shows an edge to each entity it imports through a use statement, then recurses into each of those entities' own imports the same way.

Rooted at app.js below, the graph shows app.js importing ui.js and data.js. ui.js imports components.js, and data.js imports components.js and api.js. components.js imports utils.js and validators.js; api.js imports utils.js. utils.js and validators.js each import constants.js, which imports nothing further. The inverse graph, Imported By, shows the reverse relationship — everything that imports the selected file or Rust entity, directly or indirectly.

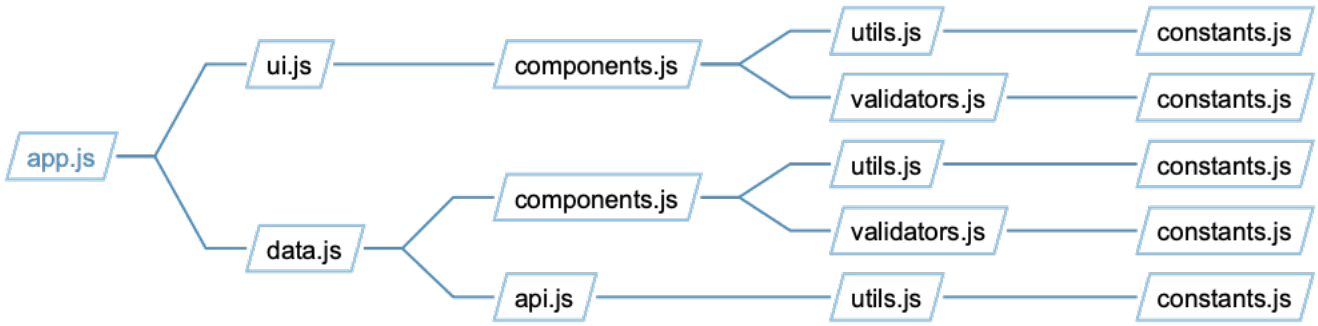
See the following code and corresponding graph

```
1 // app.js
2 import { renderPage } from './ui.js';
3 import { loadData } from './data.js';
```

```
4
5 // ui.js
6 import { createButton, createForm } from './components.js';
7
8 // data.js
9 import { createForm } from './components.js';
10 import { fetchData } from './api.js';
11
12 // components.js
13 import { buildUrl } from './utils.js';
14 import { isValidInput } from './validators.js';
15
16 // api.js
17 import { buildUrl, delay } from './utils.js';
18
19 // utils.js
20 import { API_URL, TIMEOUT } from './constants.js';
21
22 // validators.js
23 import { MAX_RETRIES } from './constants.js';
24
25 // constants.js
```

Imports - Classic

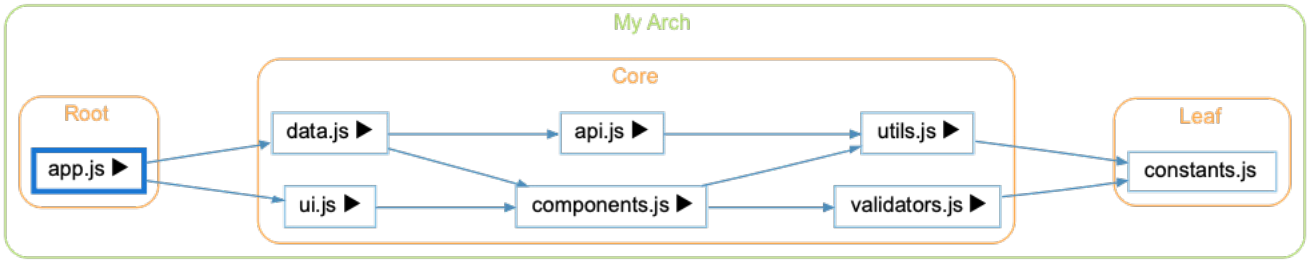
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Imports

Imports - Cluster

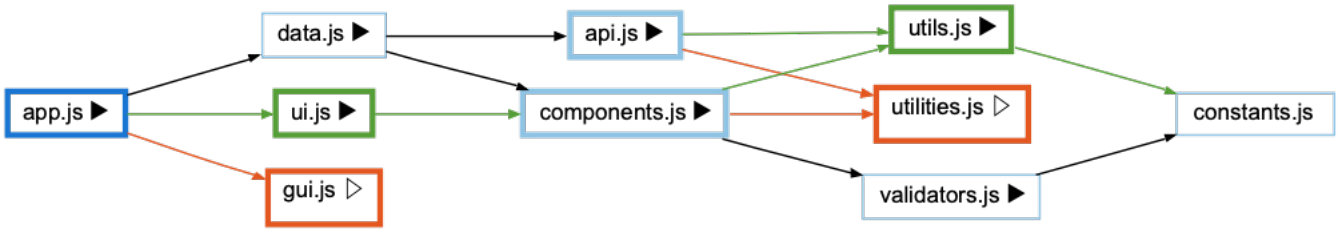
The Cluster variant groups entities by their containing class/file/architecture.



Back to Imports

Imports - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Imports

Imports - Relationship

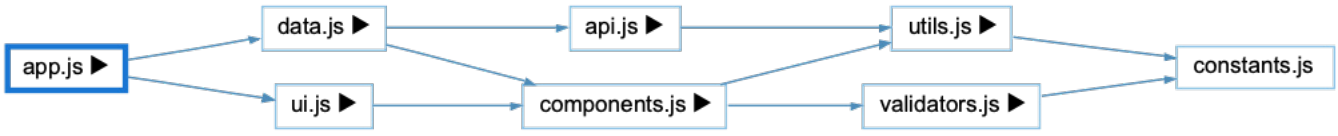
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Imports

Imports - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Imports

Imports & Imported By

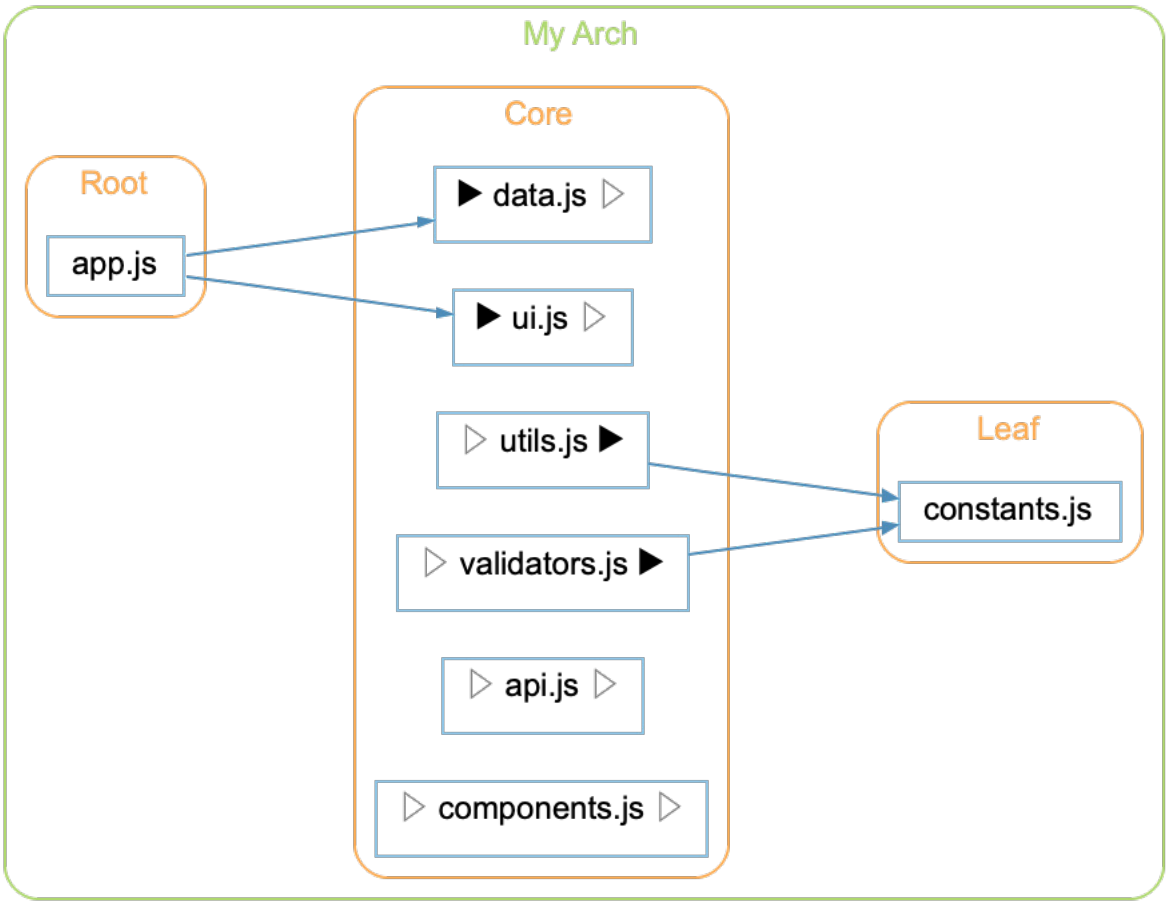
Languages: Rust, Web
Targets: Architectures
Variants: Butterfly, Imported By, Imports, Internal

Show Imports and Imported By relationships for an architecture

See Imports and Imported By graphs for details

Imports & Imported By - Butterfly

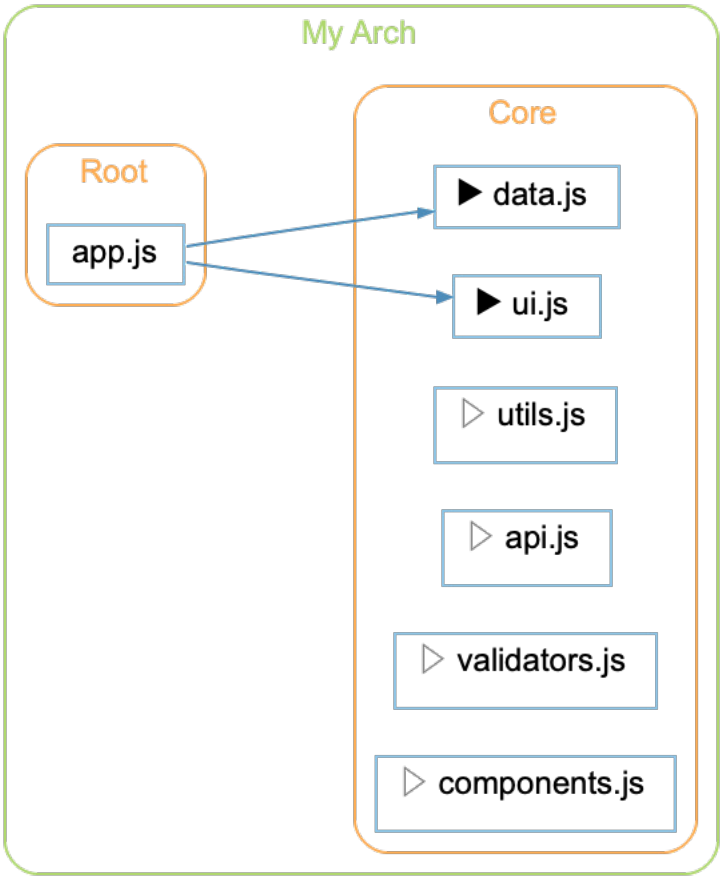
Show both Imports relationships from and Imported By relationships to entities in an architecture



Back to Imports & Imported By

Imports & Imported By - Imported By

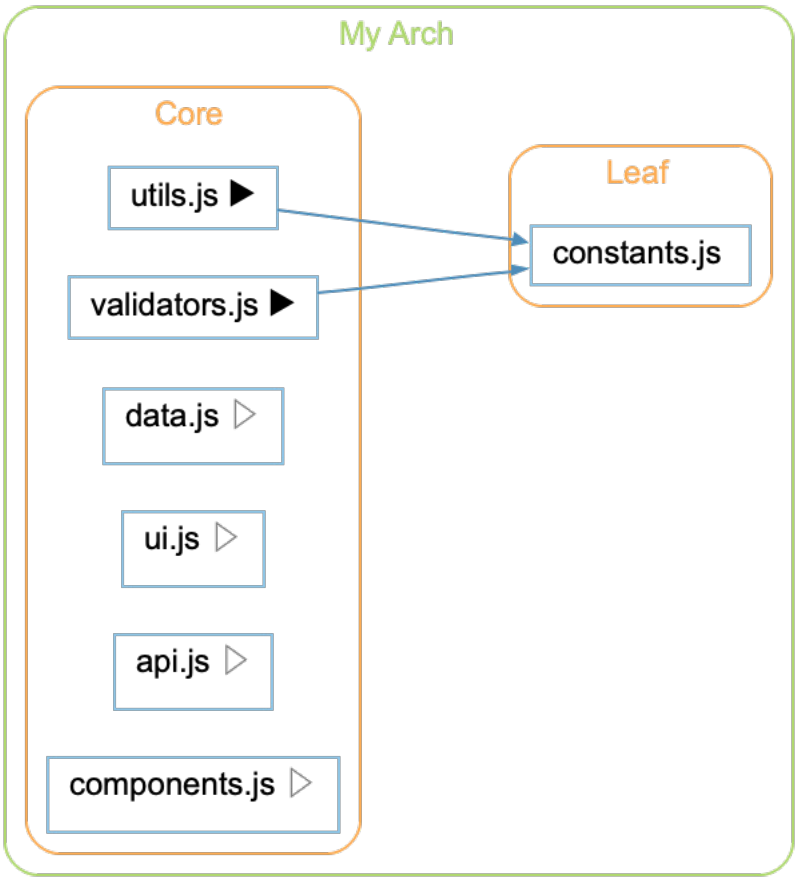
Show Imported By relationships to entities in an architecture



Back to Imports & Imported By

Imports & Imported By - Imports

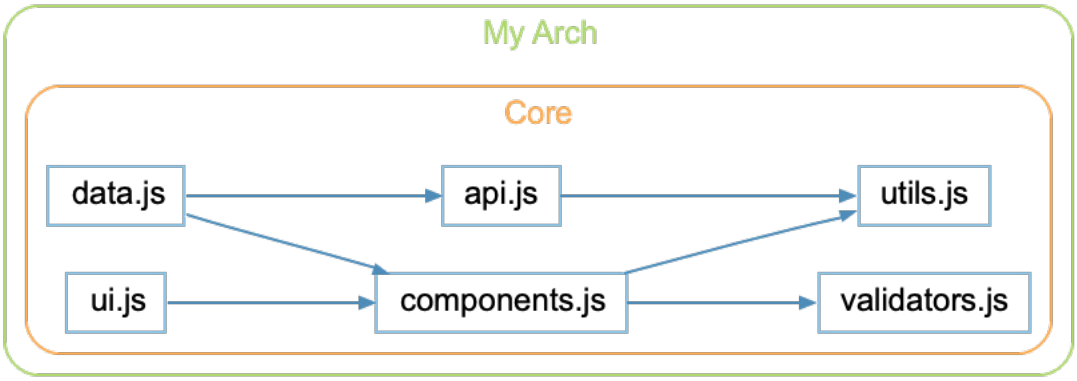
Show Imports relationships from entities in an architecture



Back to Imports & Imported By

Imports & Imported By - Internal

Show Imports relationships between entities in an architecture



Back to Imports & Imported By

Include

Languages: C++, Fortran, Pascal
Targets: Files, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show what a selected file includes, directly or indirectly.

For a selected C or Pascal file, this shows an edge to each file it includes, then recurses into each of those files' own includes, building a multi-level tree. For a selected Fortran program, function, subroutine, or block data, this shows an edge to each file it includes; Fortran's INCLUDE statement is issued by the program unit itself rather than by another file, so this doesn't extend into a deeper chain.

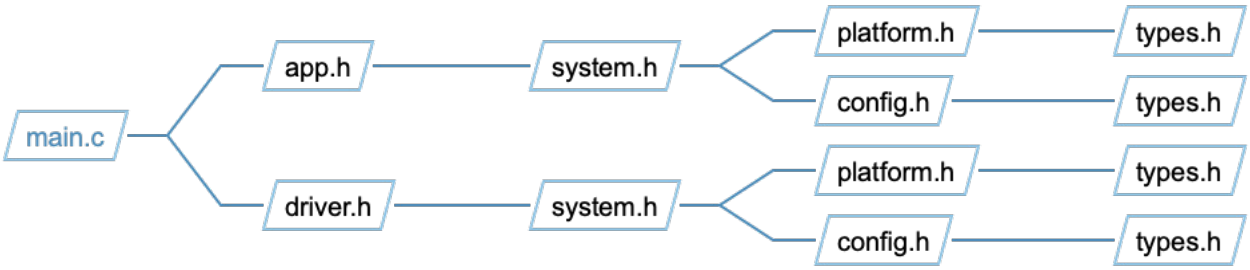
Rooted at main.c below, the graph shows main.c including app.h and driver.h. Both app.h and driver.h include system.h, which itself includes platform.h and config.h. platform.h and config.h each include types.h, which includes nothing further. The inverse graph, Includeby, shows the reverse relationship — everything that includes the selected file, directly or indirectly.

See the following code and corresponding graph

```
1 // main.c
2 #include "app.h"
3 #include "driver.h"
4
5 // app.h
6 #include "system.h"
7
8 // driver.h
9 #include "system.h"
10
11 // system.h
12 #include "platform.h"
13 #include "config.h"
14
15 // platform.h
16 #include "types.h"
17
18 // config.h
19 #include "types.h"
20
21 // types.h
```

Include - Classic

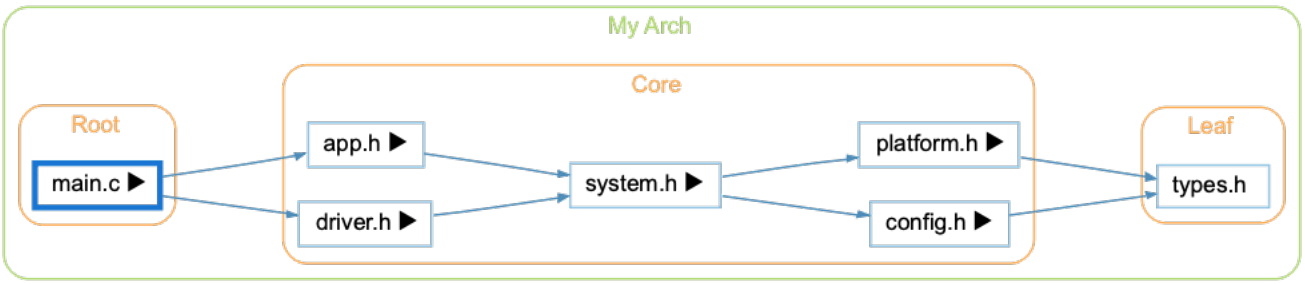
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Include

Include - Cluster

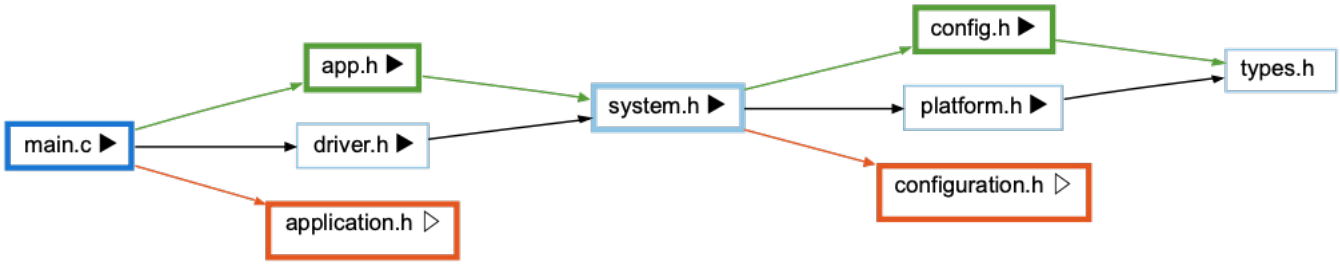
The Cluster variant groups entities by their containing class/file/architecture.



Back to Include

Include - Compare

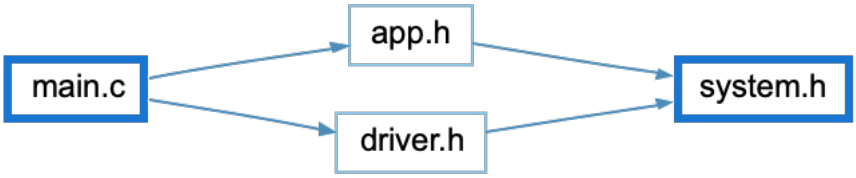
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Include

Include - Relationship

The Relationship variant filters the graph to only paths connecting the two entities.



Back to Include

Include - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Include

Include & Includeby

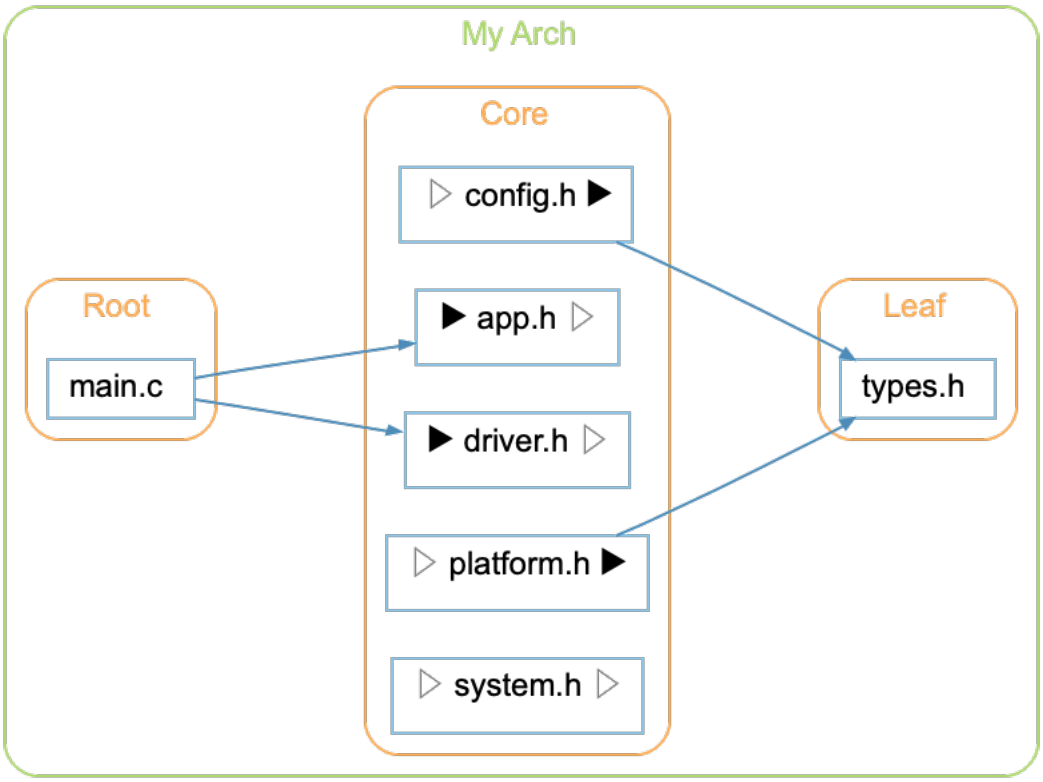
Languages: C++, Fortran, Pascal
Targets: Architectures
Variants: Butterfly, Include, Includeby, Internal

Show Include and Includeby relationships for an architecture

See Include and Includeby graphs for details

Include & Includeby - Butterfly

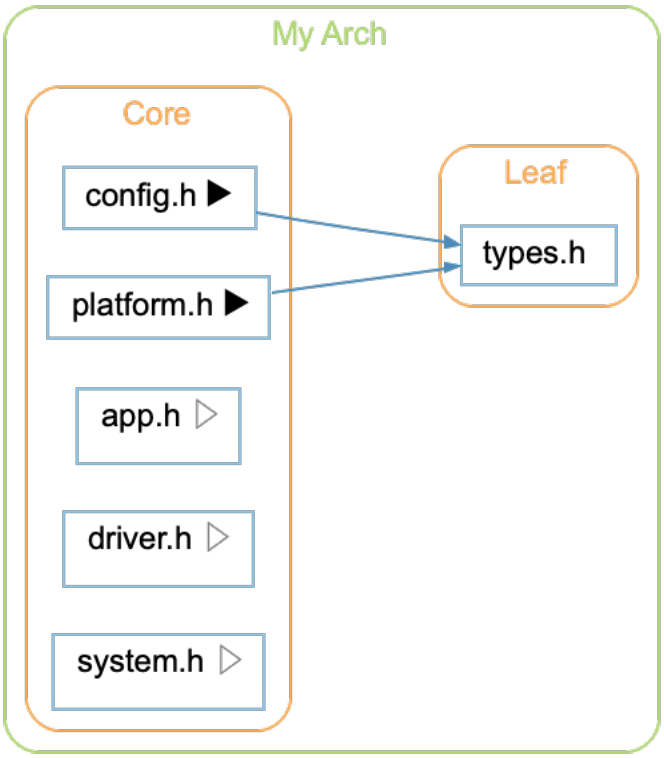
Show both Include relationships from and Includeby relationships to entities in an architecture



Back to Include & Includeby

Include & Includeby - Include

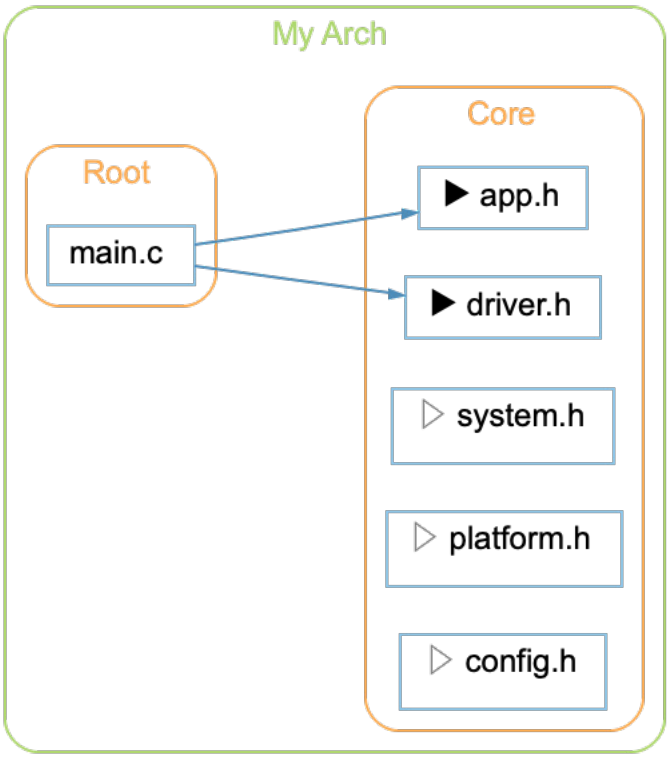
Show Include relationships from entities in an architecture



Back to Include & Includeby

Include & Includeby - Includeby

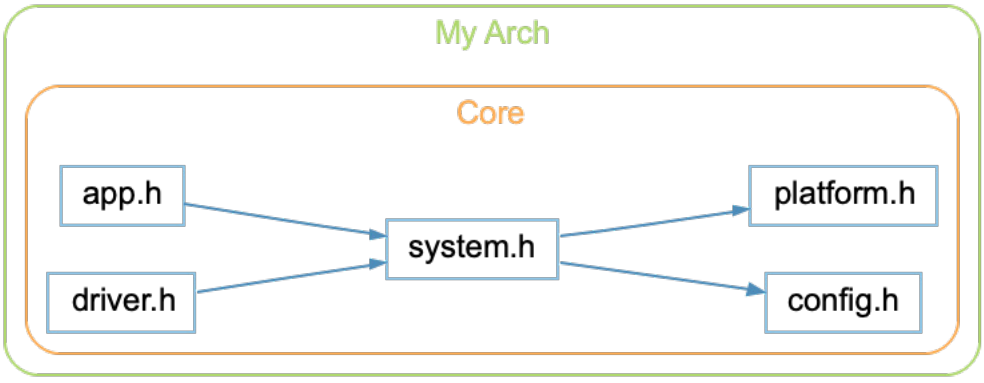
Show Includeby relationships to entities in an architecture



Back to Include & Includeby

Include & Includeby - Internal

Show Include relationships between entities in an architecture



Back to Include & Includeby

Includeby

Languages: C++, Fortran, Pascal
Targets: Files, Subprograms

Variants: Classic, Cluster, Compare, Relationship, Simplified

Show what includes a selected file, directly or indirectly.

For a selected C or Pascal file, this shows an edge to each file that includes it, then recurses into each of those files' own includers, building a multi-level tree. For a selected Fortran file, this shows an edge to each program, function, subroutine, or block data that includes it; since Fortran's INCLUDE statement is issued by the program unit itself rather than by another file, this doesn't extend into a deeper chain.

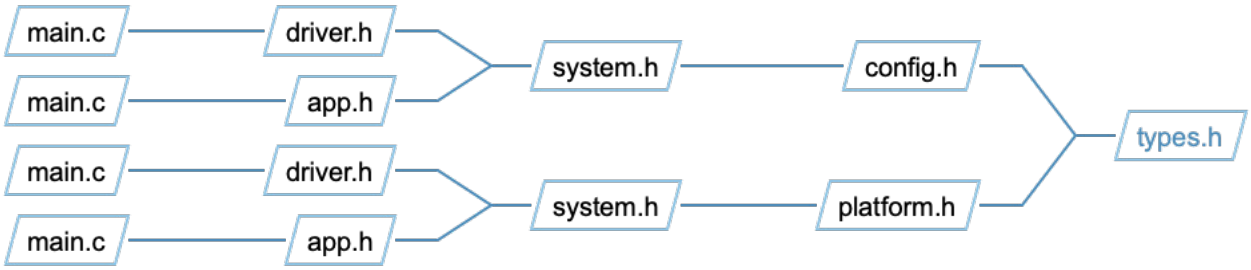
Rooted at types.h below, the graph shows types.h included by platform.h and config.h. Both are included by system.h, which is itself included by app.h and driver.h. app.h and driver.h are each included by main.c, which nothing includes. The inverse graph, Include, shows the reverse relationship — everything the selected file includes, directly or indirectly.

See the following code and corresponding graph

```
1 // main.c
2 #include "app.h"
3 #include "driver.h"
4
5 // app.h
6 #include "system.h"
7
8 // driver.h
9 #include "system.h"
10
11 // system.h
12 #include "platform.h"
13 #include "config.h"
14
15 // platform.h
16 #include "types.h"
17
18 // config.h
19 #include "types.h"
20
21 // types.h
```

Includeby - Classic

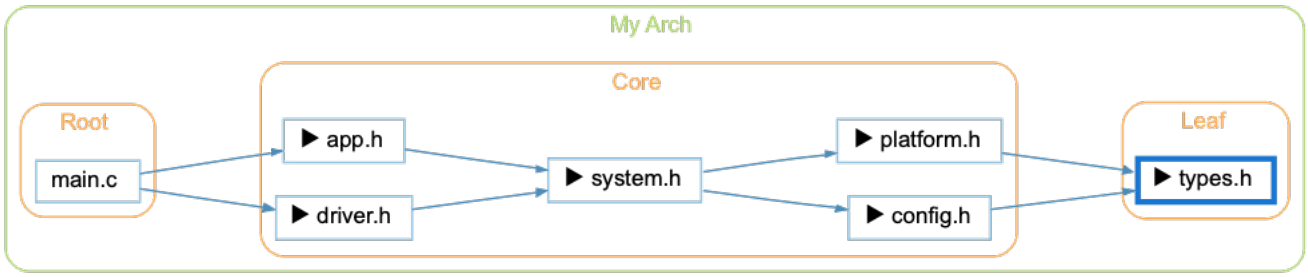
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Includeby

Includeby - Cluster

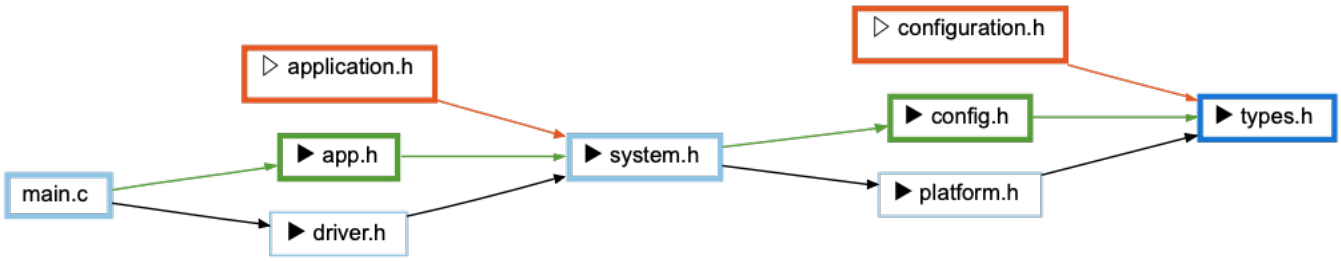
The Cluster variant groups entities by their containing class/file/architecture.



Back to Includeby

Includeby - Compare

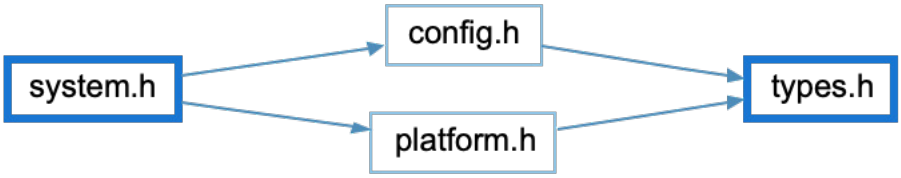
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Includeby

Includeby - Relationship

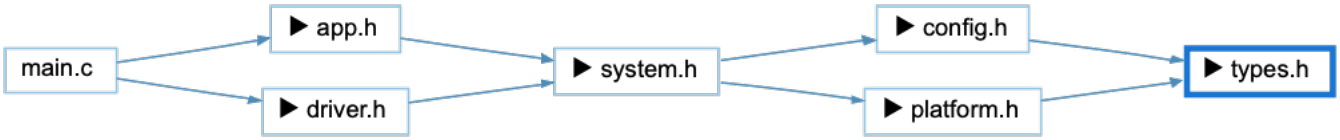
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Includeby

Includeby - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Includeby

Instantiation

Languages: VHDL
Targets: Entity Units
Variants: Custom

Show the instantiation hierarchy of a selected VHDL entity or architecture: which entities it instantiates, and which architectures implement each of those, recursively.

Each box is a specific entity/architecture pair, split into two fields — the entity's name on top, and the implementing architecture's name below. An edge leads from an architecture to each entity it instantiates, including ones instantiated inside a nested block or generate statement; enabling Show Count labels each edge with how many times that entity is instantiated. A component instantiation is resolved to whatever entity it's bound to; an entity that can't be resolved, or one that's still just an unbound component, is drawn as a gray octagon instead.

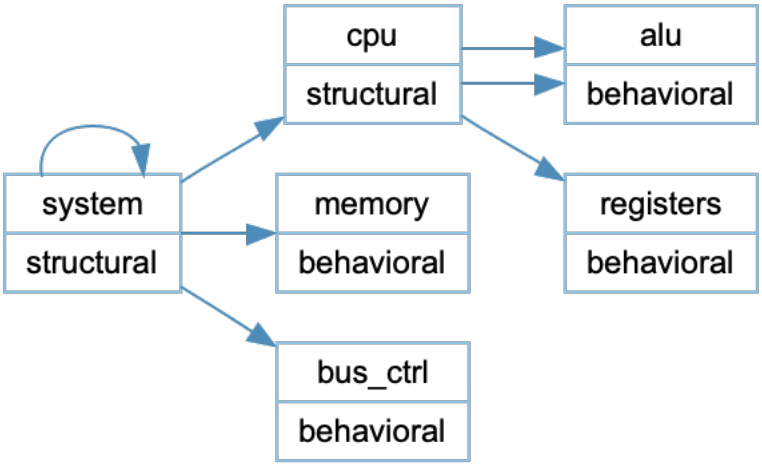
Rooted at the structural architecture of system below, the graph shows it instantiating cpu (using its structural architecture), memory, and bus_ctrl (each using their behavioral architecture). cpu's own structural architecture in turn instantiates alu and registers, both using their behavioral architecture; neither instantiates anything further, so the tree ends there.

See the following code and corresponding graph

```
1 entity system is
2   port (
```

```
3      clk : in std_logic;
4      req : in std_logic
5    );
6  end entity system;
7
8  architecture structural of system is
9    signal we      : std_logic := '0';
10   signal grant   : std_logic;
11   signal cpu_out : std_logic_vector(7 downto 0);
12   signal mem_out : std_logic_vector(7 downto 0);
13 begin
14   u_cpu : entity work.cpu(structural)
15     port map (clk => clk, we => we, din => mem_out, dout => cpu_out);
16
17   u_mem : entity work.memory(behavioral)
18     port map (clk => clk, we => we, din => cpu_out, dout => mem_out);
19
20   u_bus : entity work.bus_ctrl(behavioral)
21     port map (clk => clk, req => req, grant => grant);
22 end architecture structural;
23
24 entity cpu is
25   port (
26     clk  : in  std_logic;
27     we   : in  std_logic;
28     din  : in  std_logic_vector(7 downto 0);
29     dout : out std_logic_vector(7 downto 0)
30   );
31 end entity cpu;
32
33 architecture structural of cpu is
34   signal alu_out : std_logic_vector(7 downto 0);
35   signal reg_out : std_logic_vector(7 downto 0);
36 begin
37   u_alu : entity work.alu(behavioral)
38     port map (clk => clk, a => din, b => reg_out, result => alu_out);
39
40   u_regs : entity work.registers(behavioral)
41     port map (clk => clk, we => we, din => alu_out, dout => reg_out);
42
43   dout <= reg_out;
44 end architecture structural;
45
46 entity alu is
47   port (
48     clk    : in  std_logic;
49     a, b   : in  std_logic_vector(7 downto 0);
50     result : out std_logic_vector(7 downto 0)
51   );
52 end entity alu;
53
54 architecture behavioral of alu is
55 begin
```

```
56    -- ...
57  end architecture behavioral;
58
59  entity registers is
60    port (
61      clk  : in  std_logic;
62      we   : in  std_logic;
63      din  : in  std_logic_vector(7 downto 0);
64      dout : out std_logic_vector(7 downto 0)
65    );
66  end entity registers;
67
68  architecture behavioral of registers is
69  begin
70    -- ...
71  end architecture behavioral;
72
73  entity memory is
74    port (
75      clk  : in  std_logic;
76      we   : in  std_logic;
77      din  : in  std_logic_vector(7 downto 0);
78      dout : out std_logic_vector(7 downto 0)
79    );
80  end entity memory;
81
82  architecture behavioral of memory is
83  begin
84    -- ...
85  end architecture behavioral;
86
87  entity bus_ctrl is
88    port (
89      clk   : in  std_logic;
90      req   : in  std_logic;
91      grant : out std_logic
92    );
93  end entity bus_ctrl;
94
95  architecture behavioral of bus_ctrl is
96  begin
97    -- ...
98  end architecture behavioral;
```



This variant is a plugin. It can be enabled/disabled from the Plugin Manager.

Instantiations

Languages: Ada
Targets: Packages, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show every instantiation of a selected Ada generic package, function, procedure, or task.

For a selected generic unit, this shows an edge to each package, function, procedure, or task instantiated from it. An instantiation doesn't itself introduce a new generic to instantiate from, so this is always a single level and doesn't recurse further.

Rooted at the generic package Generic_Stack below, the graph shows edges to its three instantiations: Integer_Stack, Float_Stack, and Character_Stack.

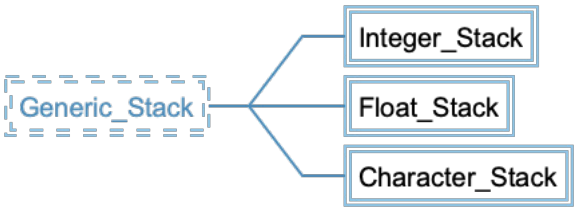
See the following code and corresponding graph

```
1 generic
2    type Element is private;
3    Capacity : Positive;
4 package Generic_Stack is
5    type Stack is limited private;
6
7    procedure Push     (S : in out Stack; Item :     Element);
8    procedure Pop      (S : in out Stack; Item : out Element);
9    function  Is_Empty (S : Stack) return Boolean;
10    function  Is_Full  (S : Stack) return Boolean;
11
12 private
13    type Element_Array is array (1 .. Capacity) of Element;
```

```
14    type Stack is record
15       Data : Element_Array;
16       Top  : Natural := 0;
17    end record;
18 end Generic_Stack;
19
20 with Generic_Stack;
21
22 package Instances is
23
24    package Integer_Stack is new Generic_Stack
25      (Element => Integer, Capacity => 64);
26
27    package Float_Stack is new Generic_Stack
28      (Element => Float, Capacity => 32);
29
30    package Character_Stack is new Generic_Stack
31      (Element => Character, Capacity => 128);
32
33 end Instances;
```

Instantiations - Classic

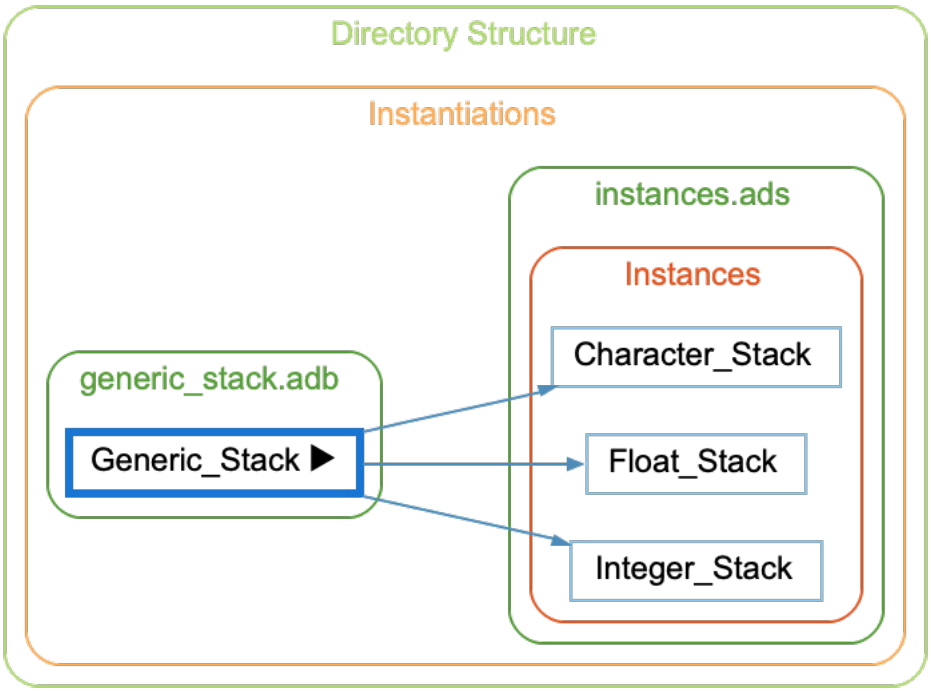
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Instantiations

Instantiations - Cluster

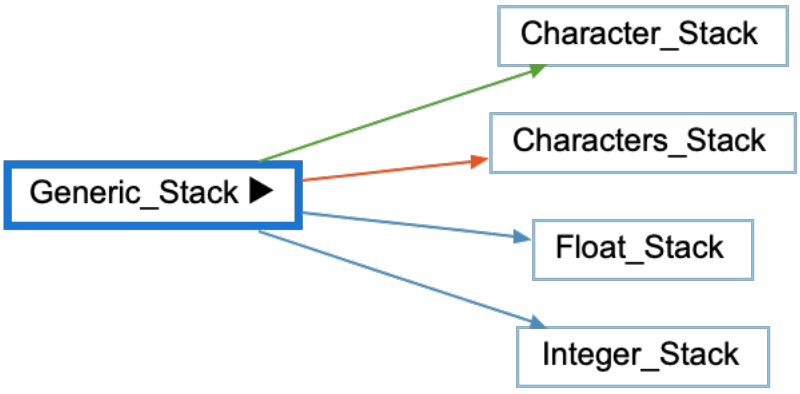
The Cluster variant groups entities by their containing class/file/architecture.



Back to Instantiations

Instantiations - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Instantiations

Instantiations - Relationship

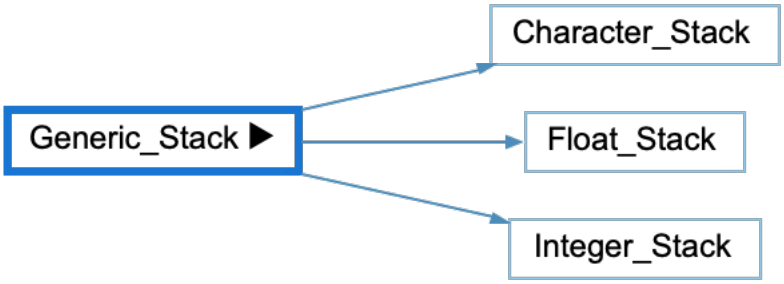
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Instantiations

Instantiations - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Instantiations

Object References

Languages: Ada, Any, C#, C++, Fortran, Java, Jovial, Pascal, Rust, VHDL, Web
Targets: Functions, Objects
Variants: Classic, Compare (Classic), With Called By (Classic), With Called By (Cluster)

Show what directly references a selected object, macro, variable, field, or property.

The root is the selected entity. Each direct reference is added as a child — anything that sets, uses, or modifies it — with the edge colored by how: a set is green, a modify is orange, and a use (or anything else) is the default blue.

Rooted at GlobalObject below, the graph shows three direct references in demo.cpp: functionPtrDemo (a use, in if (GlobalObject > n)), simpleOverridesDemo (a modify, via GlobalObject++), and complexOverridesDemo (a set, via GlobalObject = 0).

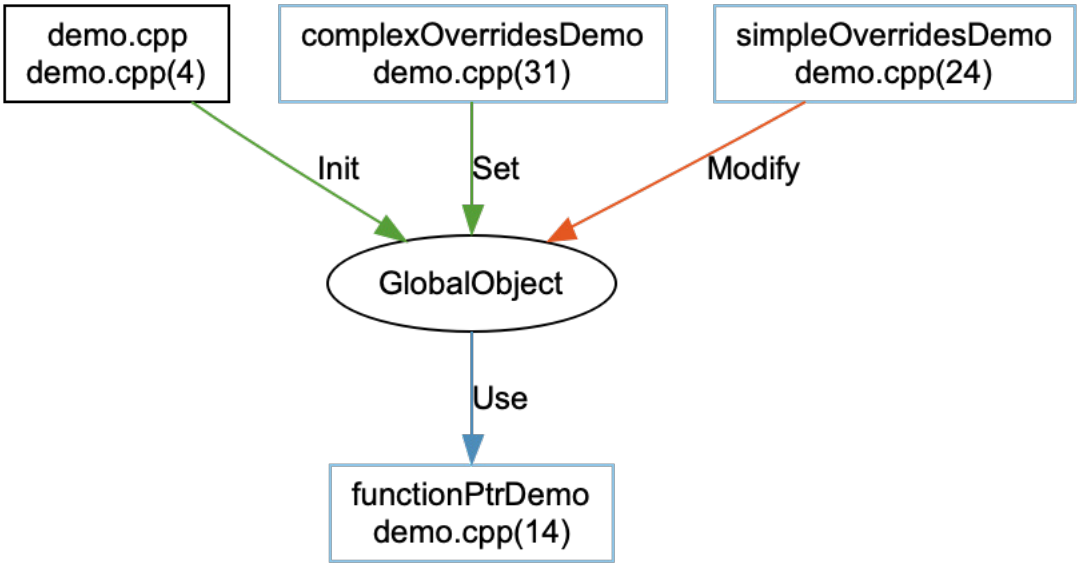
See the following code and corresponding graph

```
1 // demo.cpp
2 int GlobalObject = 0;
3
4 int functionPtrDemo(int n)
5 {
6   if (GlobalObject > n)
7     return n;
8   /* ... */
9 }
10
11 void simpleOverridesDemo()
```

```
12 {
13   /* ... */
14   GlobalObject ++;
15 }
16
17 void complexOverridesDemo()
18 {
19   /* ... */
20   GlobalObject = 0;
21 }
22
23 void callDemo(int n)
24 {
25   /* ... */
26   functionPtrDemo(n);
27   simpleOverridesDemo();
28   complexOverridesDemo();
29   /* ... */
30 }
31
32 // callby.cpp
33 class CallbyDerivedA : public CallbyBase {
34 public:
35   void pureVirtualCall() override { callDemo(0); };
36 };
37
38 void altFirst()
39 {
40   callDemo(1);
41 }
```

Object References - Classic

This variant shows only the direct references above — it doesn't continue further to callers, and doesn't cluster entities by file or architecture. See also the compare and with called by graphs.

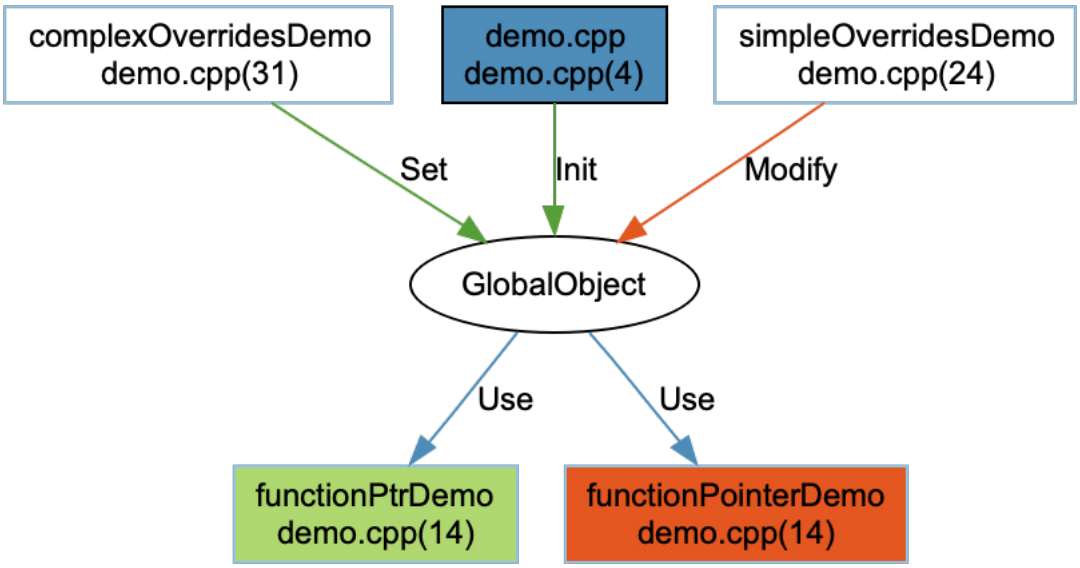


This variant is a plugin. It can be enabled/disabled from the Plugin Manager.

Back to Object References

Object References - Compare (Classic)

This variant compares those same references against the project's comparison database: an entity present unchanged in both versions is left unfilled, one whose own contents changed is filled blue, one that's new is filled green, and one that no longer exists is filled orange. In a version where functionPtrDemo was renamed from functionPointerDemo, demo.cpp is filled blue since its contents changed, and the rename shows up as two separate entities — functionPtrDemo (its new name, filled green) and functionPointerDemo (its old name, filled orange) — both still shown using GlobalObject. See also the classic and with called by graphs.

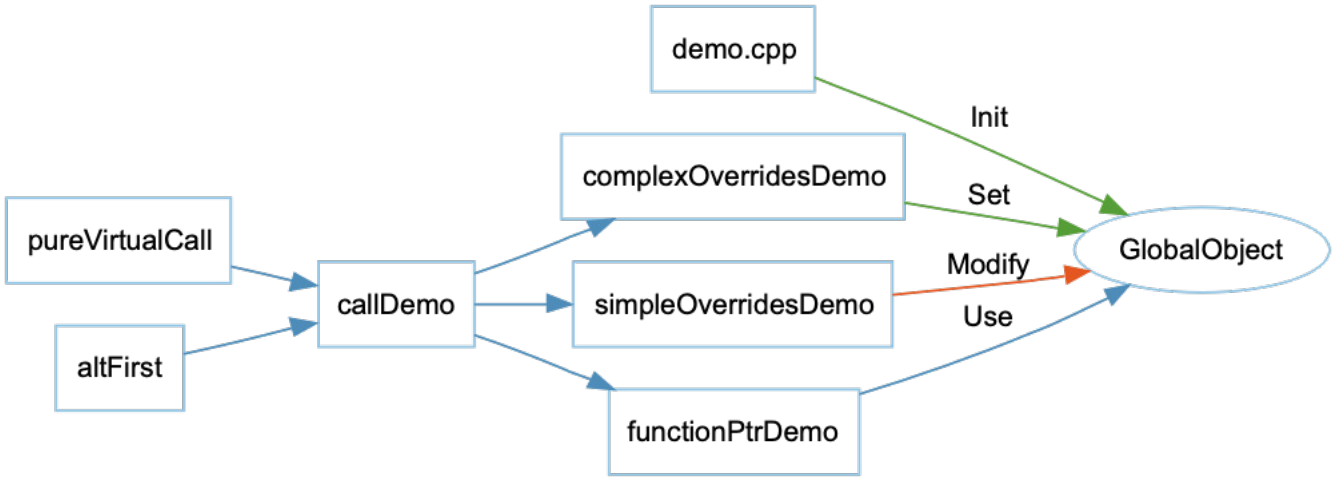


This variant is a plugin. It can be enabled/disabled from the Plugin Manager.

Back to Object References

Object References - With Called By (Classic)

This variant also continues from each reference with a Called By relationship: whatever calls that function, then whatever calls that caller in turn, as many levels deep as the Called By Depth option allows. functionPtrDemo, simpleOverridesDemo, and complexOverridesDemo are all called by callDemo, which is itself called by altFirst and by CallbyDerivedA::pureVirtualCall, both in callby.cpp. See also the classic and compare graphs.

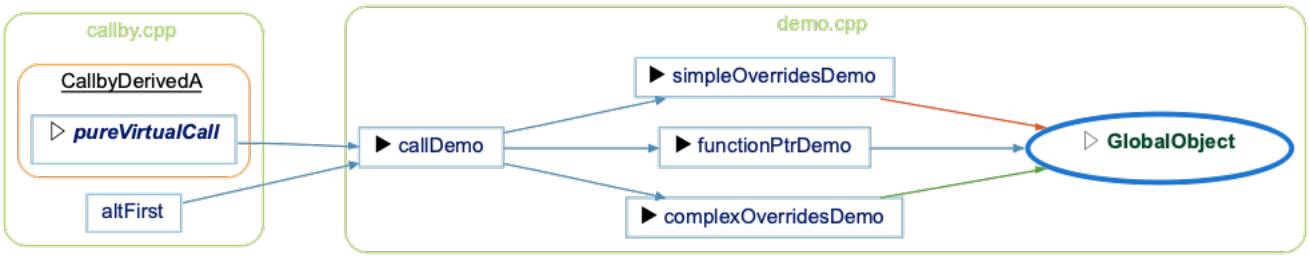


This variant is a plugin. It can be enabled/disabled from the Plugin Manager.

Back to Object References

Object References - With Called By (Cluster)

This variant also continues from each reference with a Called By relationship: whatever calls that function, then whatever calls that caller in turn, as many levels deep as the Called By Depth option allows. functionPtrDemo, simpleOverridesDemo, and complexOverridesDemo are all called by callDemo, which is itself called by altFirst and by CallbyDerivedA::pureVirtualCall, both in callby.cpp. This variant also supports clustering.



Back to Object References

Overriddenby

Languages: Basic, C#, C++
Targets: Functions
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the virtual member functions that directly or indirectly override a selected function.

For a selected virtual member function, this shows an edge to each function in a derived class that directly overrides it, then recurses downward into whatever overrides those functions in turn, building a multi-level tree that follows the class hierarchy.

Rooted at Vertebrate::describe below, the graph shows edges to Mammal::describe and Bird::describe, then continues into Dog::describe/Cat::describe and Eagle::describe/Hawk::describe, following the same descendants as the Derived Classes graph. The inverse graph, Overrides, shows the reverse relationship — the functions the selected function overrides.

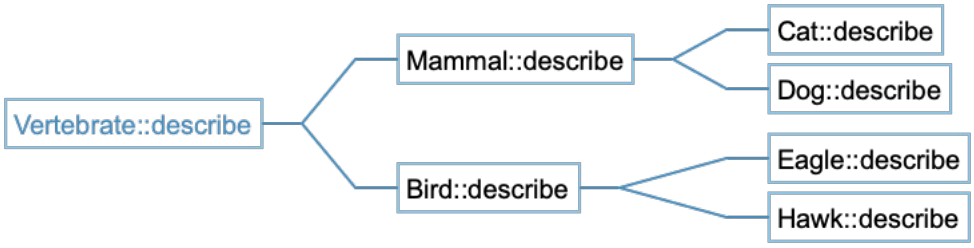
See the following code and corresponding graph

```
1 class Vertebrate {
2 public:
3     void describe() const;
4 };
5
6 class Mammal : public Vertebrate {
7 public:
8     void describe() const override;
9 };
10
11 class Dog : public Mammal {
12 public:
13     void describe() const override;
14 };
15
```

```
16 class Cat : public Mammal {
17 public:
18     void describe() const override;
19 };
20
21 class Bird : public Vertebrate {
22 public:
23     void describe() const override;
24 };
25
26 class Eagle : public Bird {
27 public:
28     void describe() const override;
29 };
30
31 class Hawk : public Bird {
32 public:
33     void describe() const override;
34 };
```

Overriddenby - Classic

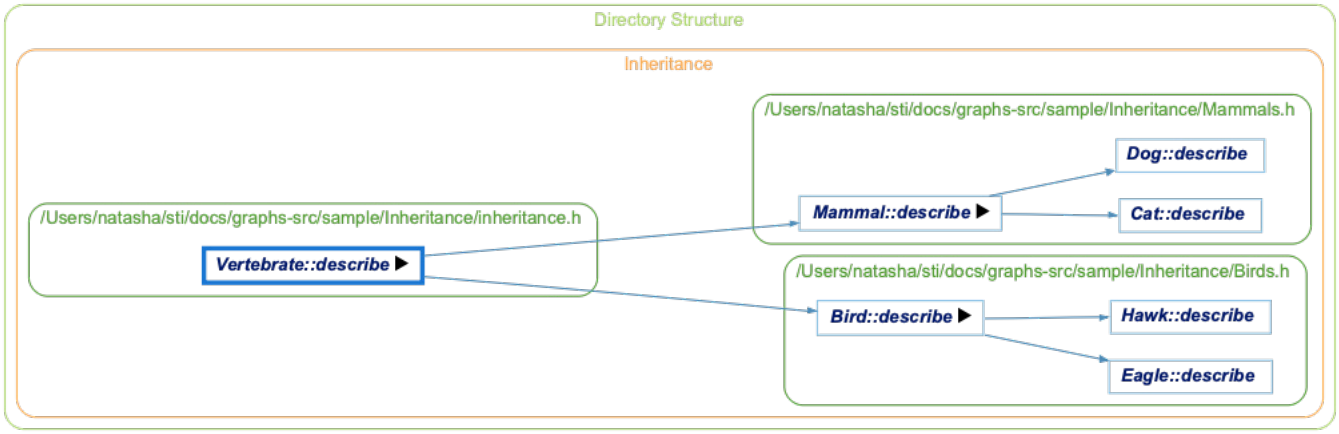
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Overriddenby

Overriddenby - Cluster

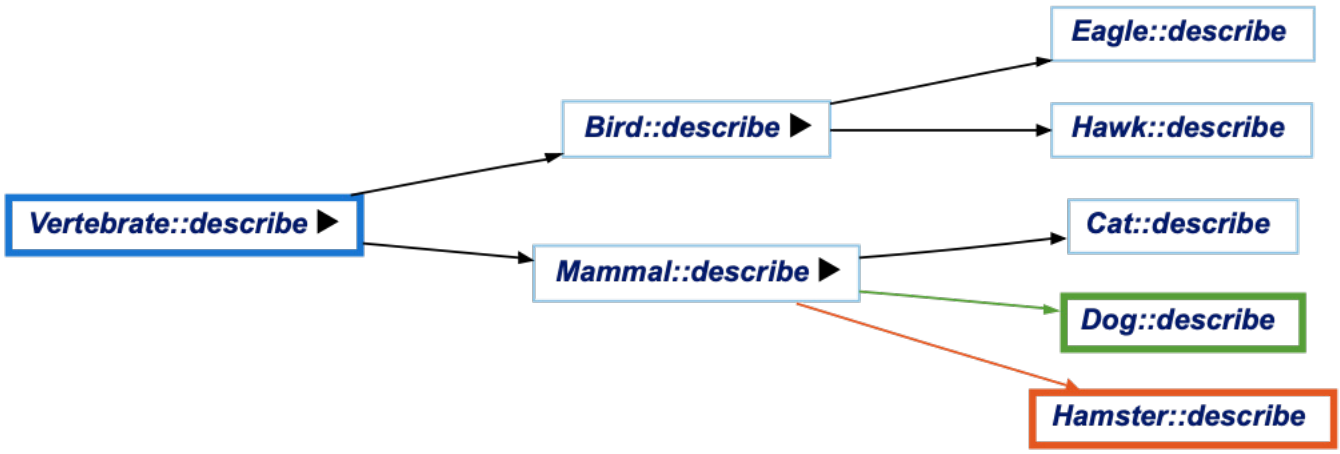
The Cluster variant groups entities by their containing class/file/architecture.



Back to Overriddenby

Overriddenby - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Overriddenby

Overriddenby - Relationship

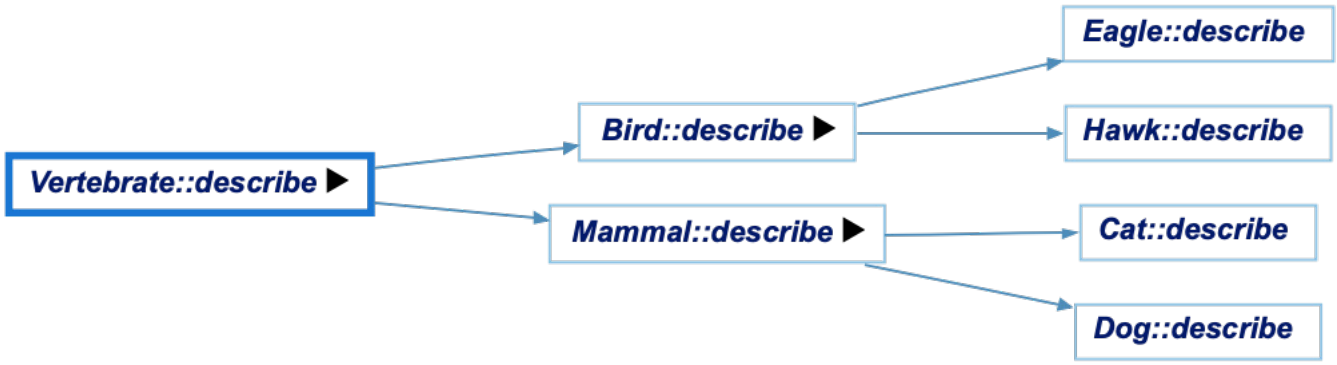
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Overriddenby

Overriddenby - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Overriddenby

Overriddenby & Overrides

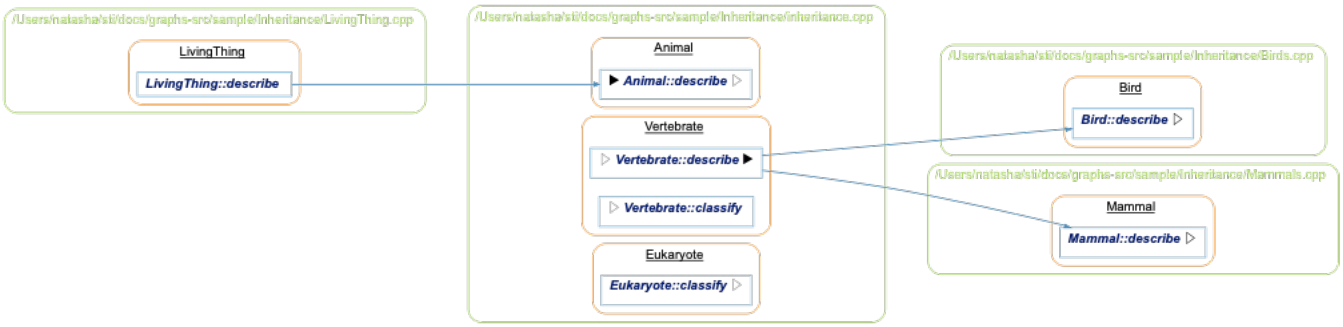
Languages: Basic, C#, C++
Targets: Architectures, Classes, Files
Variants: Butterfly, Internal, Overriddenby, Overrides

Show Overriddenby and Overrides relationships for a class, file, or architecture

See Overriddenby and Overrides graphs for details

Overriddenby & Overrides - Butterfly

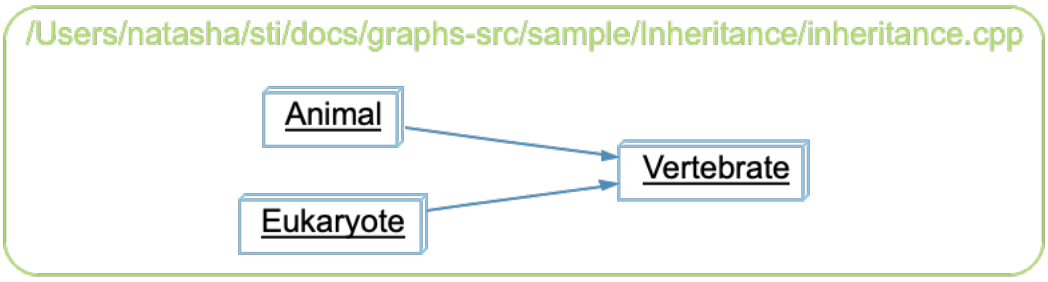
Show both Overriddenby relationships from and Overrides relationships to entities in a class, file, or architecture



Back to Overriddenby & Overrides

Overriddenby & Overrides - Internal

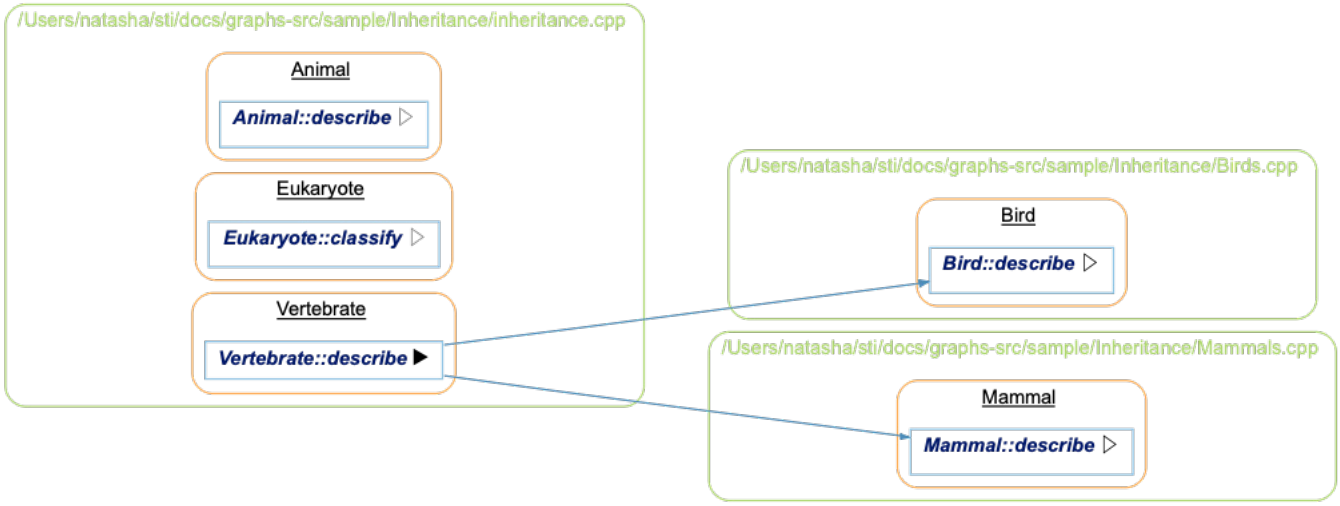
Show Overriddenby relationships between entities in a class, file, or architecture



Back to Overriddenby & Overrides

Overriddenby & Overrides - Overriddenby

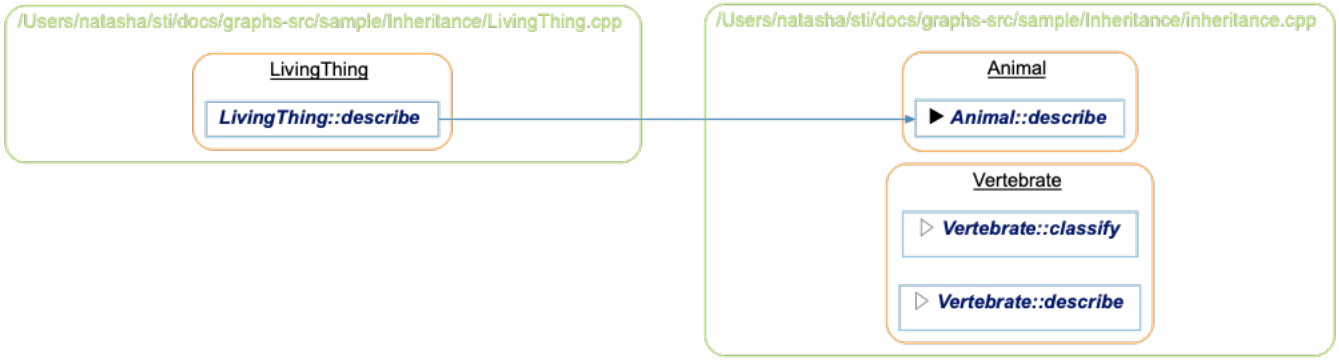
Show Overriddenby relationships from entities in a class, file, or architecture



Back to Overriddenby & Overrides

Overriddenby & Overrides - Overrides

Show Overrides relationships to entities in a class, file, or architecture



Back to Overriddenby & Overrides

Overrides

Languages: Basic, C#, C++
Targets: Functions
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the virtual member functions a selected function directly or indirectly overrides.

For a selected virtual member function, this shows an edge to the function it directly overrides in its base class, then recurses upward into whatever that function itself overrides, building a multi-level tree that follows the class hierarchy.

Rooted at Vertebrate::describe below, the graph shows an edge to Animal::describe, then continues up to LivingThing::describe, following the same ancestry as the Base Classes graph. The inverse graph, Overridden By, shows the reverse relationship — the functions that override the selected function.

See the following code and corresponding graph

```
1 class LivingThing {
2 public:
3     virtual void describe() const;
4 };
5
6 class Animal : public LivingThing {
7 public:
8     void describe() const override;
9 };
10
11 class Vertebrate : public Animal {
12 public:
13     void describe() const override;
14 };
```

Overrides - Classic

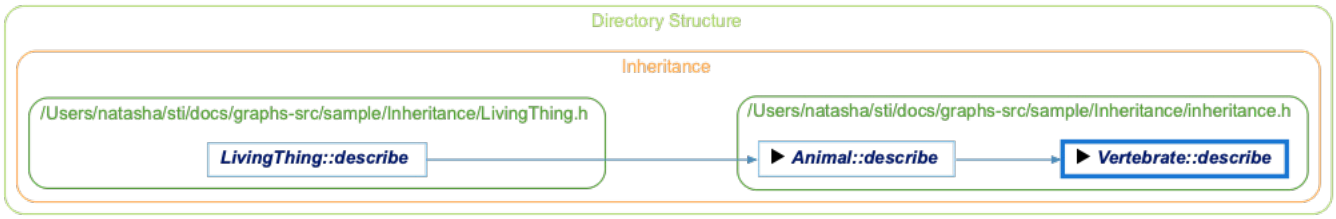
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Overrides

Overrides - Cluster

The Cluster variant groups entities by their containing class/file/architecture.



Back to Overrides

Overrides - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Overrides

Overrides - Relationship

The Relationship variant filters the graph to only paths connecting the two entities.



Back to Overrides

Overrides - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Overrides

Parent Declaration

Languages: Ada, C++, Fortran, VHDL, Web
Targets: Classes, Functions, Modules, Objects, Packages, Subprograms, Types
Variants: Classic

Show the Declaration graph for the scope a selected entity is declared inside.

For a selected Ada function, procedure, package, task, entry, component, enumeration literal, constant, exception, object, or type, this jumps to its enclosing package, subprogram, task, or protected object. For a selected PHP or JavaScript function, property, parameter, constant, variable, class, or interface, this jumps to the file or class it's defined in. For a selected parameter, object, subroutine, type, or variable in other languages, this jumps to its enclosing compool, file, function, method, procedure, subroutine, or type. For a selected C or C++ struct, class, or union member, or enum enumerator, this jumps to its enclosing type. In each of these cases, the graph shown is that scope's own Declaration graph.

For a selected VHDL configuration, entity, function, package, or procedure, this instead jumps to its enclosing package or entity and shows that scope's Declaration Tree — the same tree shown when browsing that parent directly.

For example, the Parent Declaration graph for Compute (declared inside the Ada package Declarations) is the Declaration graph for Declarations, shown in Declaration - Ada. Likewise, the Parent Declaration graph for area (a method of the PHP class Shape) is the Declaration graph for Shape, shown in Declaration - Web Class.

Parent Lib Unit

Languages: Ada
Targets: Packages, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the Ada package a selected library unit is declared under, and its own parent packages in turn.

For a selected package, function, or procedure declared with dotted (child unit) notation, this shows an edge to the package it's declared under, then recurses upward into that package's own parent, building a multi-level ancestry tree. Only a package can have children, so a function or procedure is always the bottom of this chain, never itself a parent.

Rooted at Telemetry.Sensors.Analog below, the graph shows an edge to Telemetry.Sensors, then continues up to the top-level package Telemetry. The inverse graph, Child Lib Units, shows the reverse relationship — everything declared under the selected package.

See the following code and corresponding graph

```
1 package Telemetry is
2 end Telemetry;
3
4 package Telemetry.Sensors is
5 end Telemetry.Sensors;
6
7 package Telemetry.Sensors.Analog is
8    Max_Channels : constant := 8;
9 end Telemetry.Sensors.Analog;
```

Parent Lib Unit - Classic

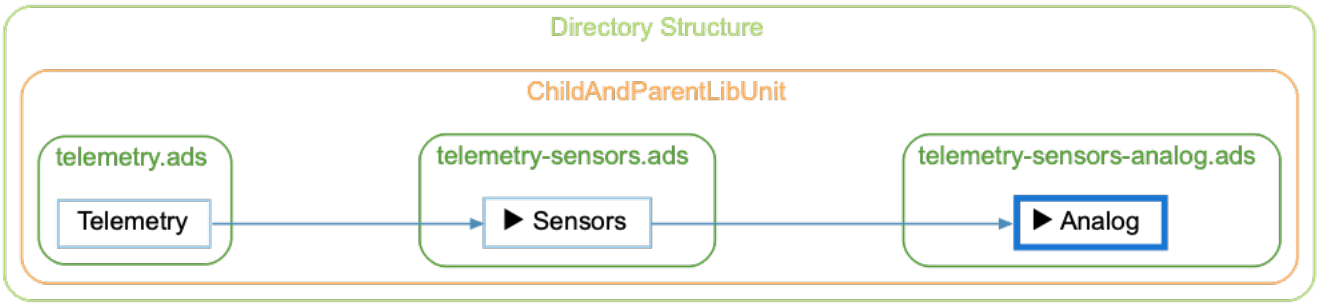
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Parent Lib Unit

Parent Lib Unit - Cluster

The Cluster variant groups entities by their containing class/file/architecture.



Back to Parent Lib Unit

Parent Lib Unit - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Parent Lib Unit

Parent Lib Unit - Relationship

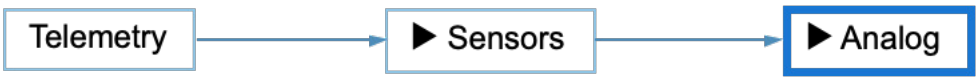
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Parent Lib Unit

Parent Lib Unit - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Parent Lib Unit

Rename Declaration

Languages: Ada
Targets: Objects, Packages, Subprograms, Types
Variants: Classic

Show the Declaration graph for the entity a selected Ada renaming declaration refers to.

For a selected function, procedure, package, task, entry, component, enumeration literal, constant, exception, object, or type declared as a rename of another entity (an Ada renames declaration), this jumps to the renamed entity and shows its own Declaration graph.

For example, the Rename Declaration graph for Run (declared inside Main as procedure Run (Count : Integer; Label : String) renames Client.Run;) is the Declaration graph for Client.Run, shown in Declaration - Ada.

Required By

Languages: Rust, Web
Targets: Files, Generics
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show what requires a selected JavaScript file or Rust trait, directly or indirectly.

For a selected JavaScript file, this shows an edge to each file that requires it, then recurses into each of those files' own requirers, building a multi-level tree. For a selected Rust trait, this shows an edge to each generic parameter, associated type, or trait that requires it as a bound, then continues to recurse if one of those requirers is itself a trait with its own requirers.

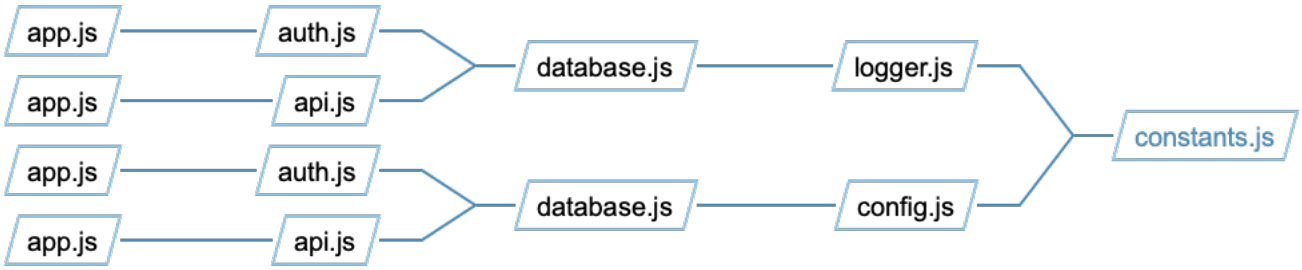
Rooted at constants.js below, the graph shows constants.js required by config.js and logger.js. Both are required by database.js, which is itself required by api.js and auth.js. api.js and auth.js are each required by app.js, which nothing requires. The inverse graph, Requires, shows the reverse relationship — everything the selected file or Rust entity requires, directly or indirectly.

See the following code and corresponding graph

```
// app.js
const { getUsers } = require('./api');
const { login } = require('./auth');

// api.js
const { connect, query } = require('./database');

// auth.js
const { connect, query } = require('./database');

// database.js
const { log, error } = require('./logger');
const { config } = require('./config');

// config.js
const { API_URL, MAX_RETRIES } = require('./constants');

// logger.js
const { TIMEOUT_MS } = require('./constants');

// constants.js
```

Required By - Classic

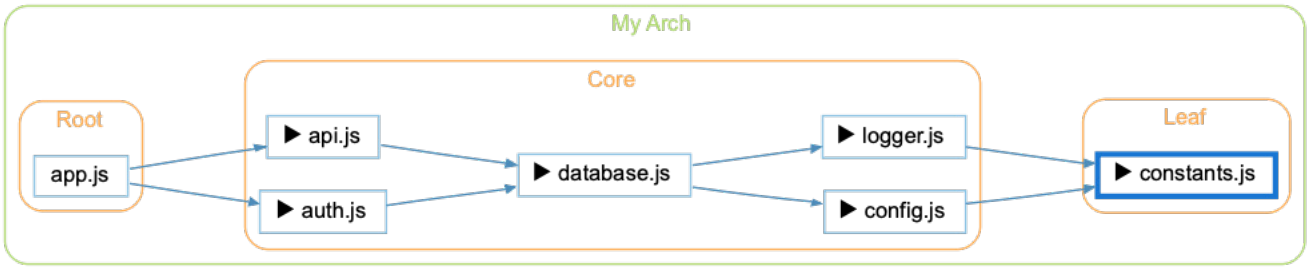
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Required By

Required By - Cluster

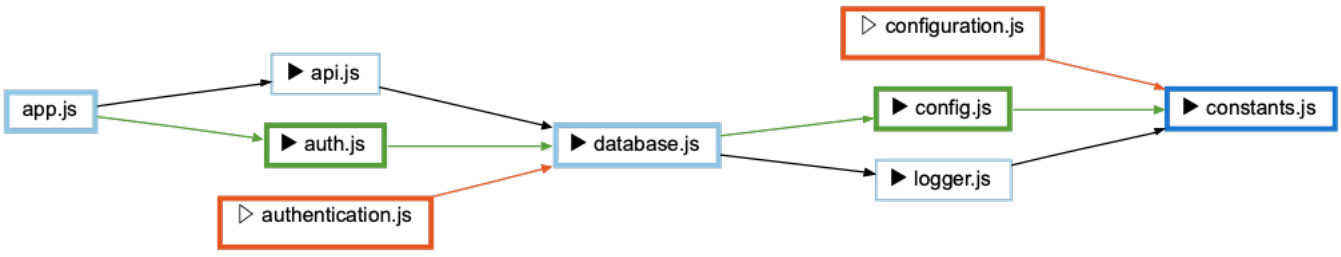
The Cluster variant groups entities by their containing class/file/architecture.



Back to Required By

Required By - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Required By

Required By - Relationship

The Relationship variant filters the graph to only paths connecting the two entities.



Back to Required By

Required By - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Required By

Requires

Languages: Rust, Web
Targets: Files, Generics
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show what a selected JavaScript file or Rust generic parameter, associated type, or trait requires, directly or indirectly.

For a selected JavaScript file, this shows an edge to each module it requires, then recurses into each of those files' own requires, building a multi-level tree. For a Rust generic parameter, associated type, or trait, this shows an edge to each trait it requires as a bound; since a trait can itself require another trait as a supertrait bound, this can continue to recurse.

Rooted at app.js below, the graph shows app.js requiring api.js and auth.js. Both api.js and auth.js require database.js, which itself requires logger.js and config.js. logger.js and config.js each require constants.js, which requires nothing further. The inverse graph, Required By, shows the reverse relationship — everything that requires the selected file or Rust entity, directly or indirectly.

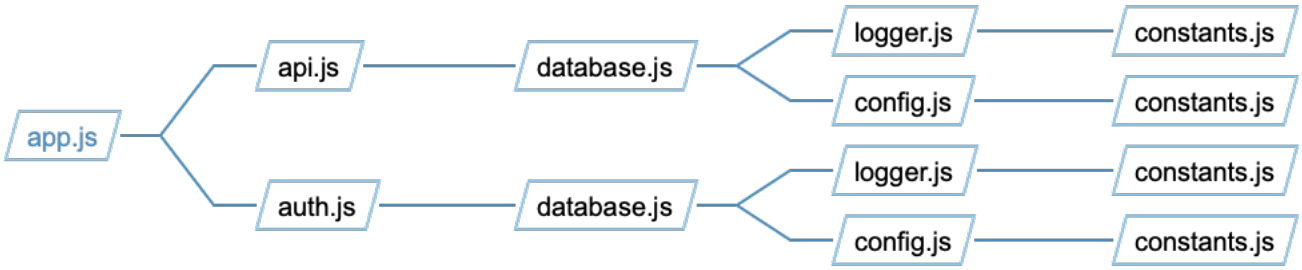
See the following code and corresponding graph

```
1 // app.js
2 const { getUsers } = require('./api');
3 const { login } = require('./auth');
4
5 // api.js
```

```
6 const { connect, query } = require('./database');
7
8 // auth.js
9 const { connect, query } = require('./database');
10
11 // database.js
12 const { log, error } = require('./logger');
13 const { config } = require('./config');
14
15 // config.js
16 const { API_URL, MAX_RETRIES } = require('./constants');
17
18 // logger.js
19 const { TIMEOUT_MS } = require('./constants');
20
21 // constants.js
```

Requires - Classic

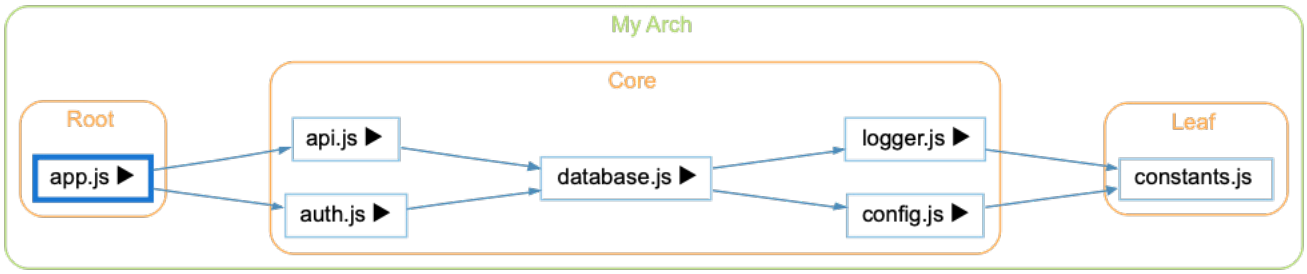
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Requires

Requires - Cluster

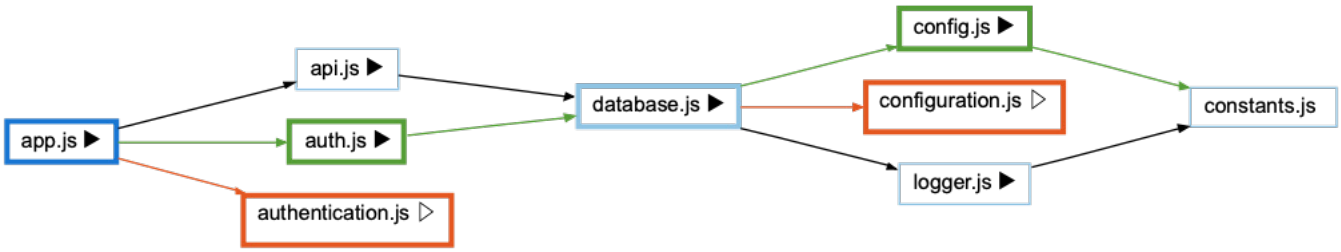
The Cluster variant groups entities by their containing class/file/architecture.



Back to Requires

Requires - Compare

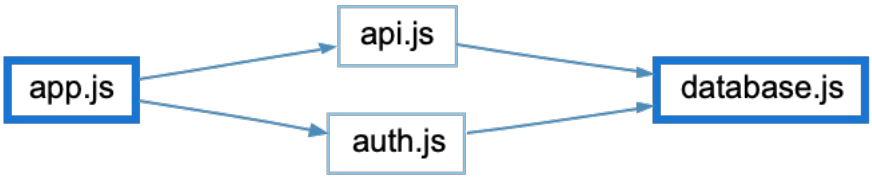
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Requires

Requires - Relationship

The Relationship variant filters the graph to only paths connecting the two entities.



Back to Requires

Requires - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Requires

Requires & Required By

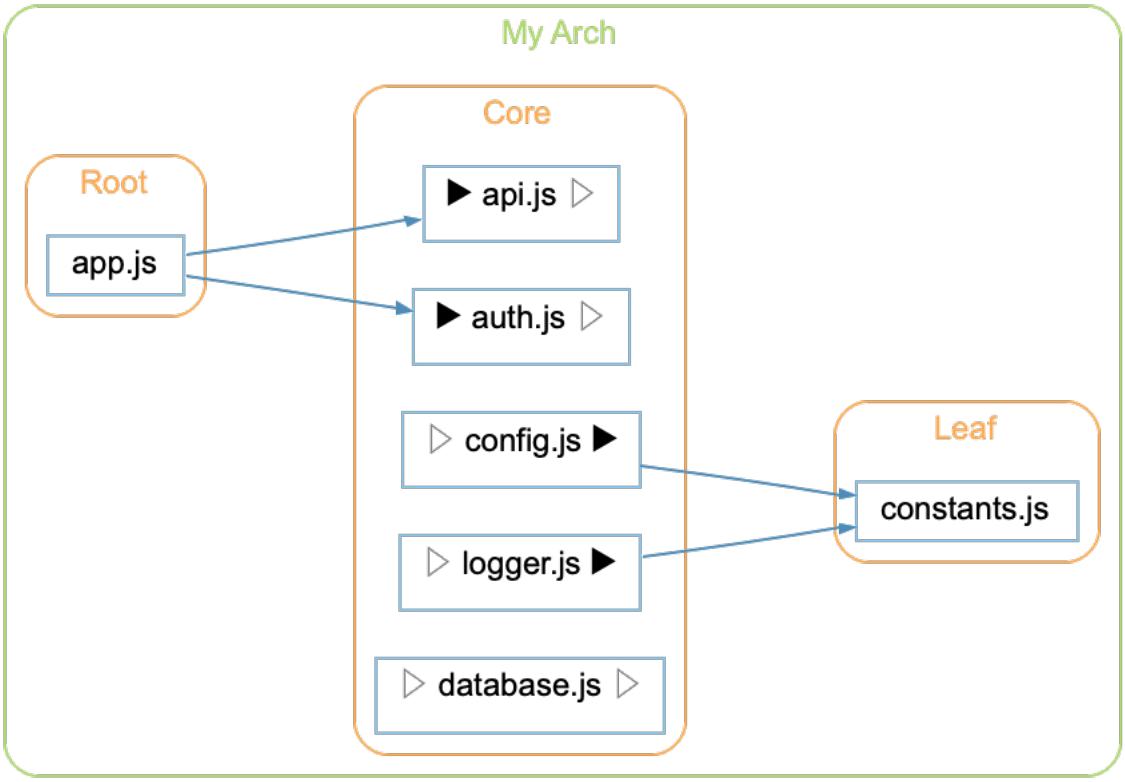
Languages: Rust, Web
Targets: Architectures
Variants: Butterfly, Internal, Required By, Requires

Show Requires and Required By relationships for an architecture

See Requires and Required By graphs for details

Requires & Required By - Butterfly

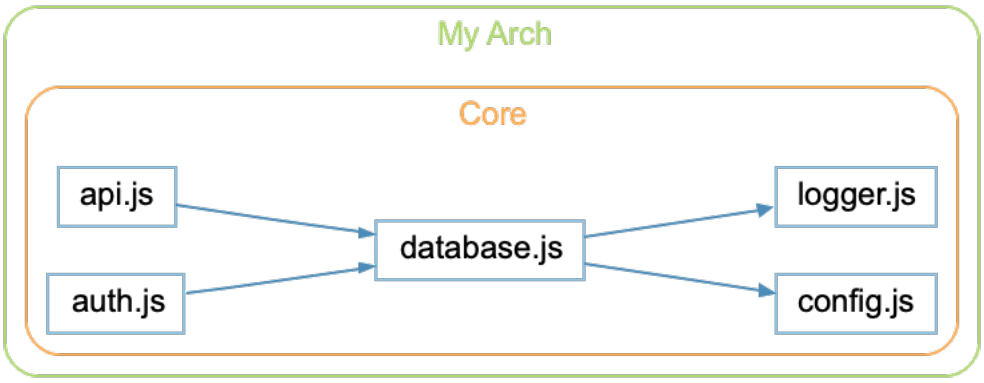
Show both Requires relationships from and Required By relationships to entities in an architecture



Back to Requires & Required By

Requires & Required By - Internal

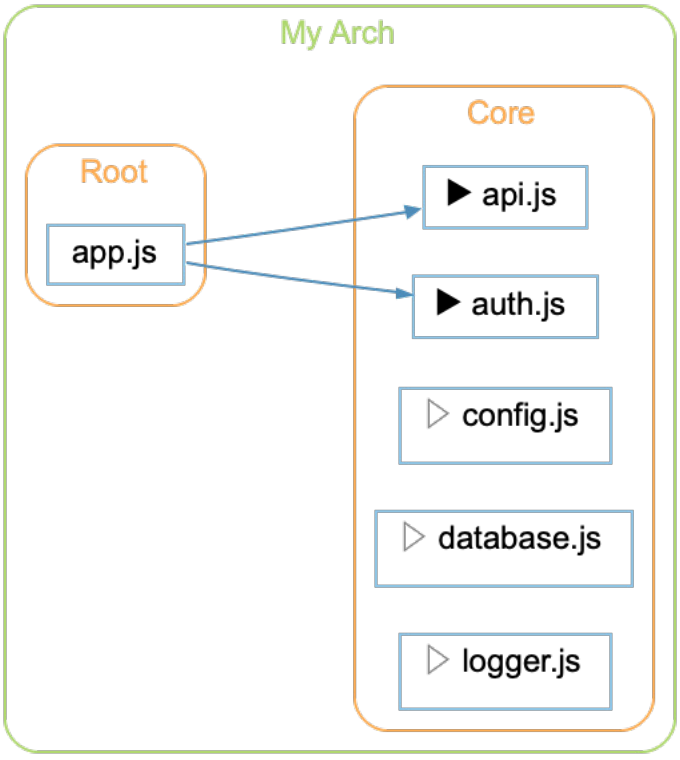
Show Requires relationships between entities in an architecture



Back to Requires & Required By

Requires & Required By - Required By

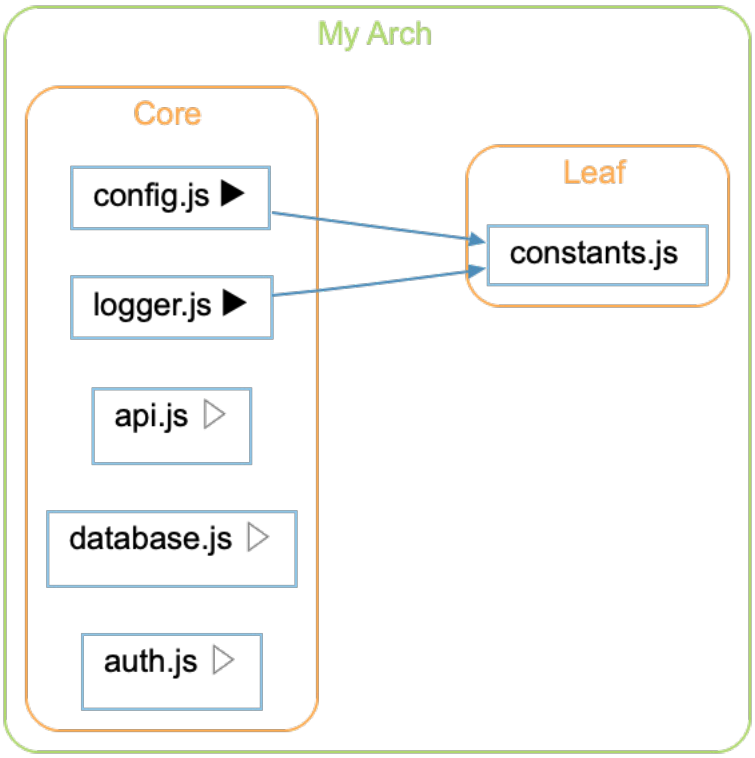
Show Required By relationships to entities in an architecture



Back to Requires & Required By

Requires & Required By - Requires

Show Requires relationships from entities in an architecture



Back to Requires & Required By

Return Type

Languages: C#, C++, Java, Jovial
Targets: Functions, Subprograms
Variants: Classic

Show the Declaration graph for a selected function's return type.

For a selected C, C#, Java, or Jovial function or method, this jumps to its return type and shows that type's own Declaration graph.

For example, the Return Type graph for color (declared as Color color() const; in the C++ class Shape) is the Declaration graph for the enum Color, shown in Declaration - Enum.

Type Derived From

Languages: Ada
Targets: Types
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the Ada types a selected type is directly or indirectly derived from.

For a selected Ada tagged type, this shows an edge to the type it directly derives from, then recurses upward into that type's own parent, building a multi-level ancestry chain.

Rooted at Calibrated_Sensor below, the graph shows an edge to its parent, Analog_Sensor, then continues up to Sensor. The inverse graph, Type Tree, shows the reverse relationship — everything derived from the selected type, directly or indirectly.

See the following code and corresponding graph

```
1 package Sensors is
2
3    type Sensor is tagged record
4       Id     : Positive := 1;
5       Active : Boolean  := False;
6    end record;
7
8 end Sensors;
9
10 with Sensors;
11
12 package Analog is
13
14    type Analog_Sensor is new Sensors.Sensor with record
15       Raw_Value : Float := 0.0;
16       Min_Range : Float := 0.0;
17       Max_Range : Float := 1.0;
18    end record;
19
20 end Analog;
21
22 with Analog;
23
24 package Calibrated is
25
26    type Calibrated_Sensor is new Analog.Analog_Sensor with record
27       Offset : Float := 0.0;
28       Scale  : Float := 1.0;
29    end record;
30
31 end Calibrated;
```

Type Derived From - Classic

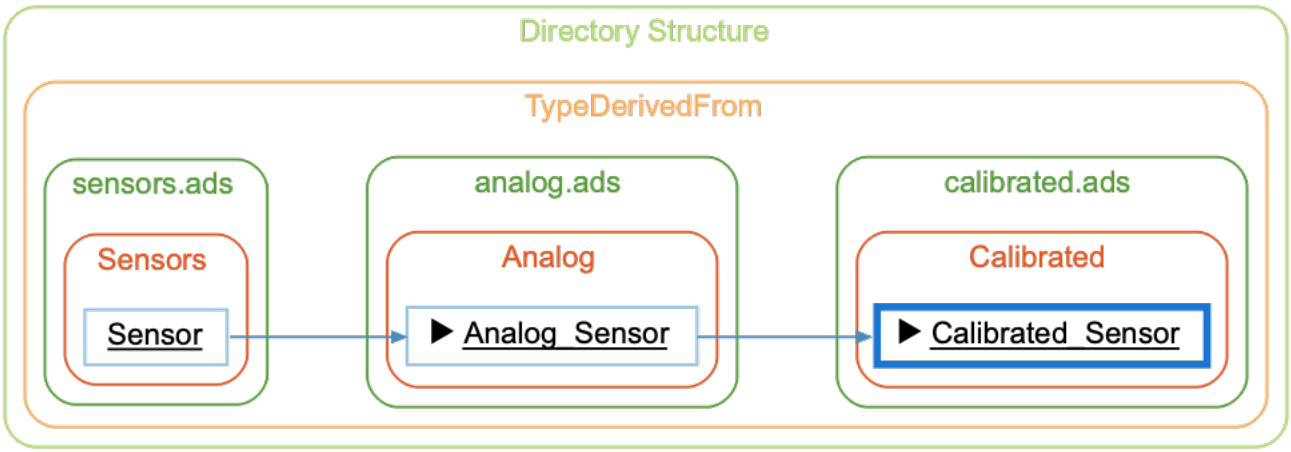
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Type Derived From

Type Derived From - Cluster

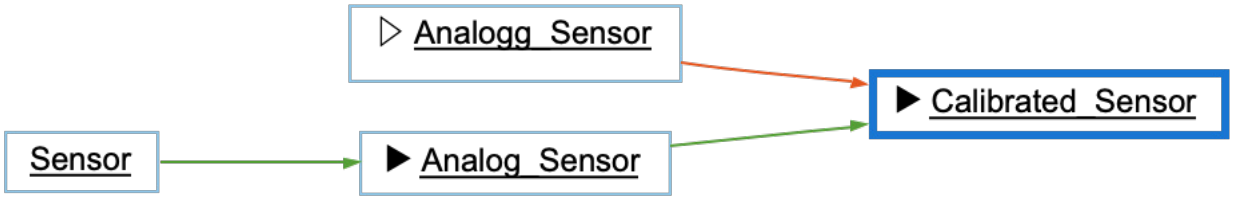
The Cluster variant groups entities by their containing class/file/architecture.



Back to Type Derived From

Type Derived From - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Type Derived From

Type Derived From - Relationship

The Relationship variant filters the graph to only paths connecting the two entities.



Back to Type Derived From

Type Derived From - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Type Derived From

Type Tree

Languages: Ada
Targets: Types
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the Ada types directly or indirectly derived from a selected type.

For a selected Ada tagged type, this shows an edge to each type that directly derives from it, then recurses into each of those types' own descendants, building a multi-level tree.

Rooted at Calibrated_Sensor below, the graph shows edges to its two direct descendants, Temperature_Sensor and Pressure_Sensor. Temperature_Sensor continues to Thermocouple, and Pressure_Sensor continues to Barometer. The inverse graph, Type Derived From, shows the reverse relationship — everything the selected type derives from, directly or indirectly.

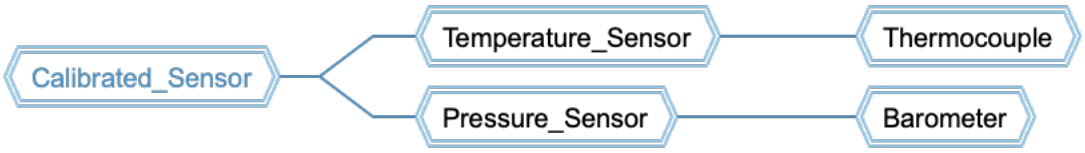
See the following code and corresponding graph

```
1 with Analog;
2
3 package Calibrated is
4
5    type Calibrated_Sensor is new Analog.Analog_Sensor with record
6       Offset : Float := 0.0;
7       Scale  : Float := 1.0;
8    end record;
9
10 end Calibrated;
11
12 with Calibrated;
13
14 package Temperature is
15
16    type Temperature_Sensor is
```

```
17       new Calibrated.Calibrated_Sensor with record
18       Unit : Character := 'C';
19    end record;
20
21    type Thermocouple is new Temperature_Sensor with record
22       Junction_Type : Character := 'K';
23    end record;
24
25 end Temperature;
26
27 with Calibrated;
28
29 package Pressure is
30
31    type Pressure_Sensor is
32       new Calibrated.Calibrated_Sensor with record
33       Unit : Character := 'P';
34    end record;
35
36    type Barometer is new Pressure_Sensor with record
37       Altitude_Compensation : Float := 0.0;
38    end record;
39
40 end Pressure;
```

Type Tree - Classic

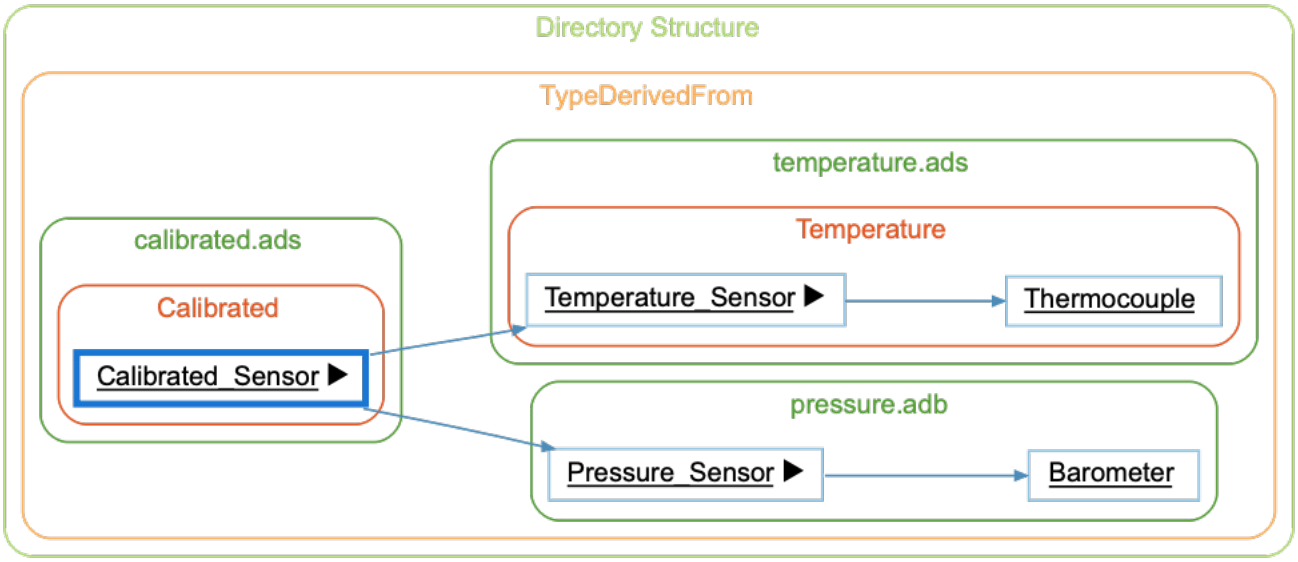
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Type Tree

Type Tree - Cluster

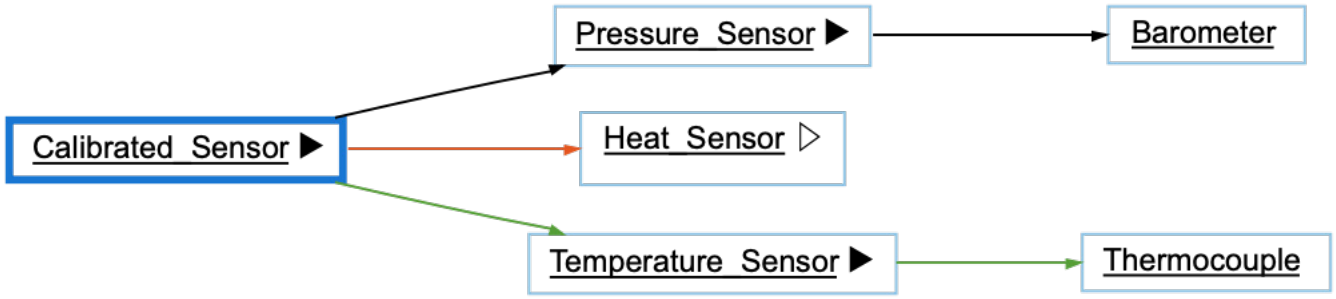
The Cluster variant groups entities by their containing class/file/architecture.



Back to Type Tree

Type Tree - Compare

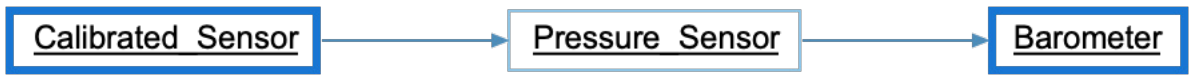
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Type Tree

Type Tree - Relationship

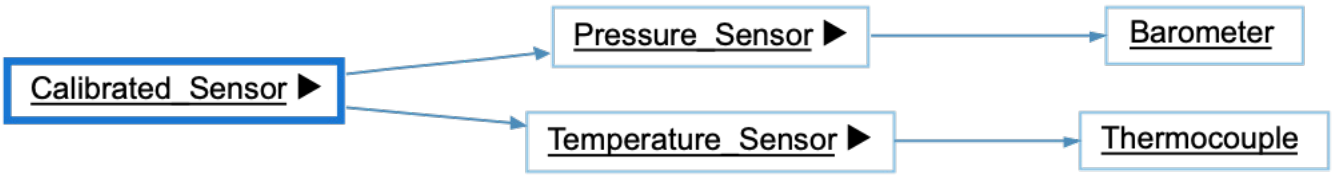
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Type Tree

Type Tree - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Type Tree

Type Tree & Type Derived From

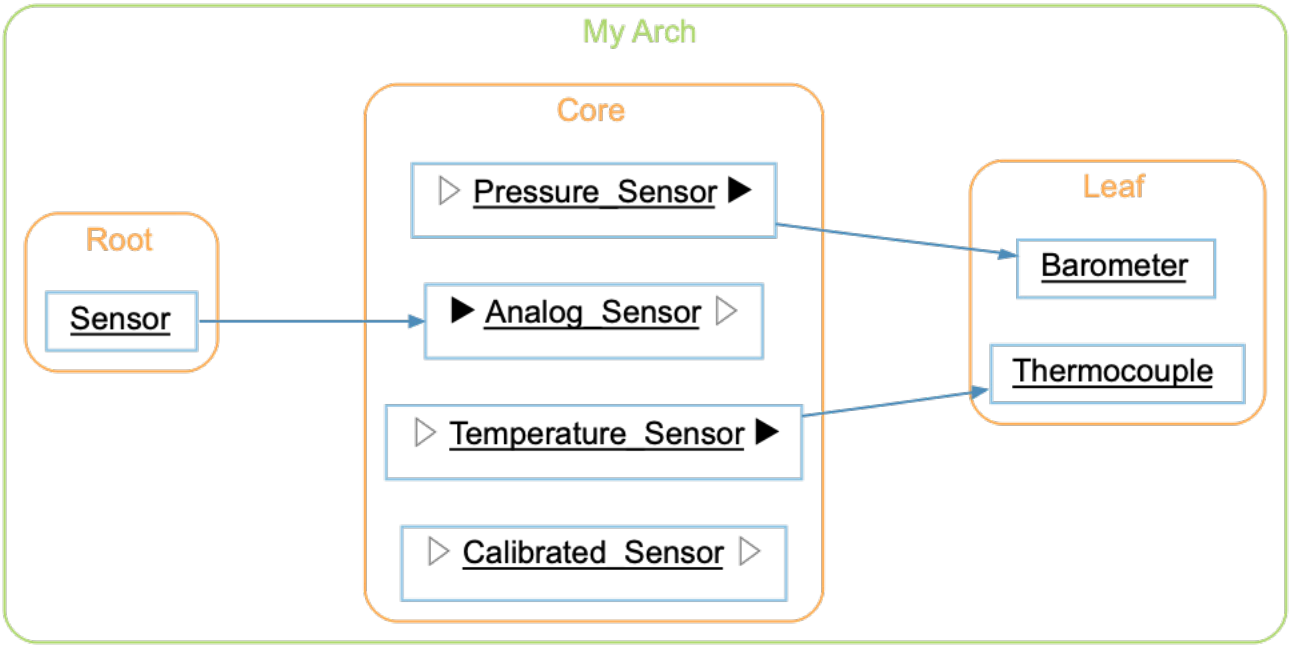
Languages: Ada
Targets: Architectures
Variants: Butterfly, Internal, Type Derived From, Type Tree

Show Type Tree and Type Derived From relationships for an architecture

See Type Tree and Type Derived From graphs for details

Type Tree & Type Derived From - Butterfly

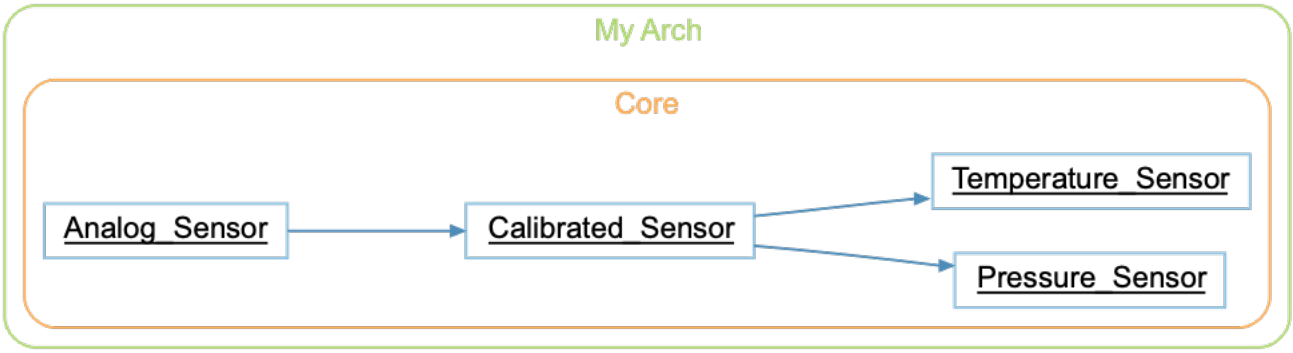
Show both Type Tree relationships from and Type Derived From relationships to entities in an architecture



Back to Type Tree & Type Derived From

Type Tree & Type Derived From - Internal

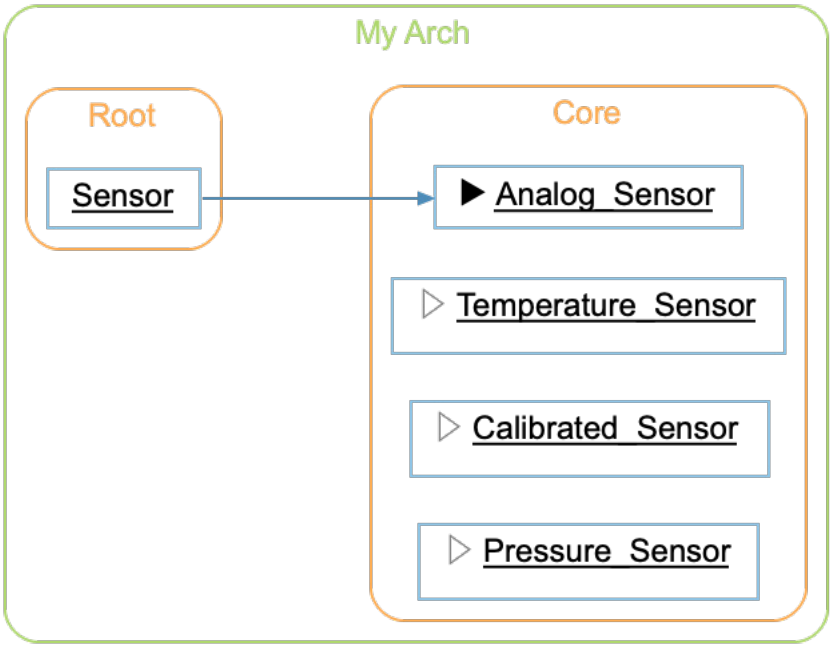
Show Type Tree relationships between entities in an architecture



Back to Type Tree & Type Derived From

Type Tree & Type Derived From - Type Derived From

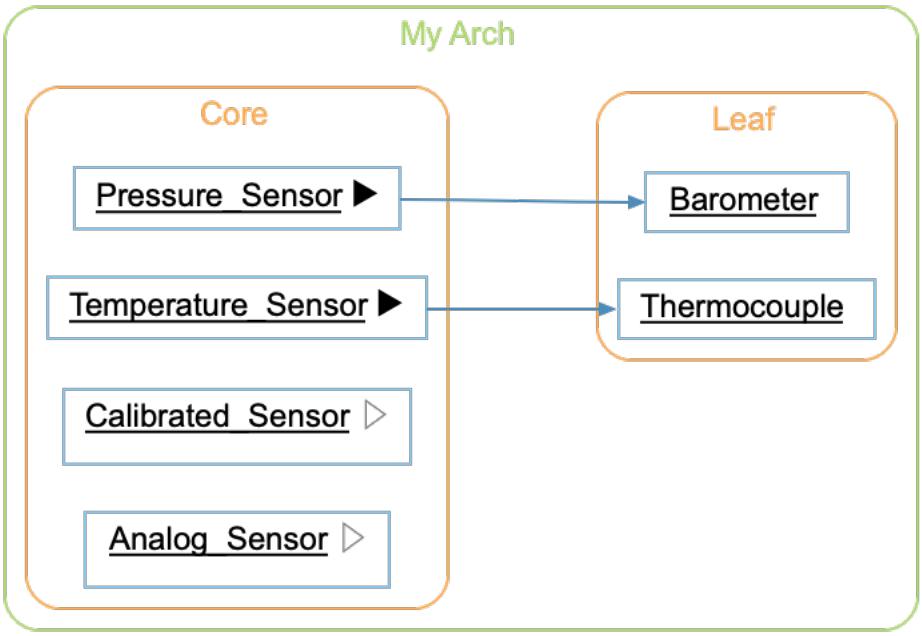
Show Type Derived From relationships to entities in an architecture



Back to Type Tree & Type Derived From

Type Tree & Type Derived From - Type Tree

Show Type Tree relationships from entities in an architecture



Back to Type Tree & Type Derived From

UML Class Diagram

Languages: Ada, C#, C++, Java, Python, Web
Targets: Architectures, Classes, Files, Packages, Record Types
Variants: Classic

Show a UML-style class diagram of a selected class, along with its base classes and subclasses.

For a class, show the UML-style class diagram of the selected class. For a file, architecture, or Python package, show the UML-style class diagram of all classes contained in it.

The main box lists the class's own fields and methods in UML notation — +/#/- prefixes for public/protected/private, name: type for fields (with an initializer after = when there is one), and name(params) returnType for methods. Its immediate base classes are added above the main box and its immediate subclasses below, connected by a hollow-triangle inheritance arrow pointing from subclass to base class. This only follows class inheritance (extends, or the equivalent in other languages) — it doesn't follow interface implementation, so a class's implemented interfaces never appear here.

See the Declaration graph for a more detailed view of the same class — it groups members the same way, but also includes nested types, and adds the interfaces a class implements (or, for an interface, the classes that implement it).

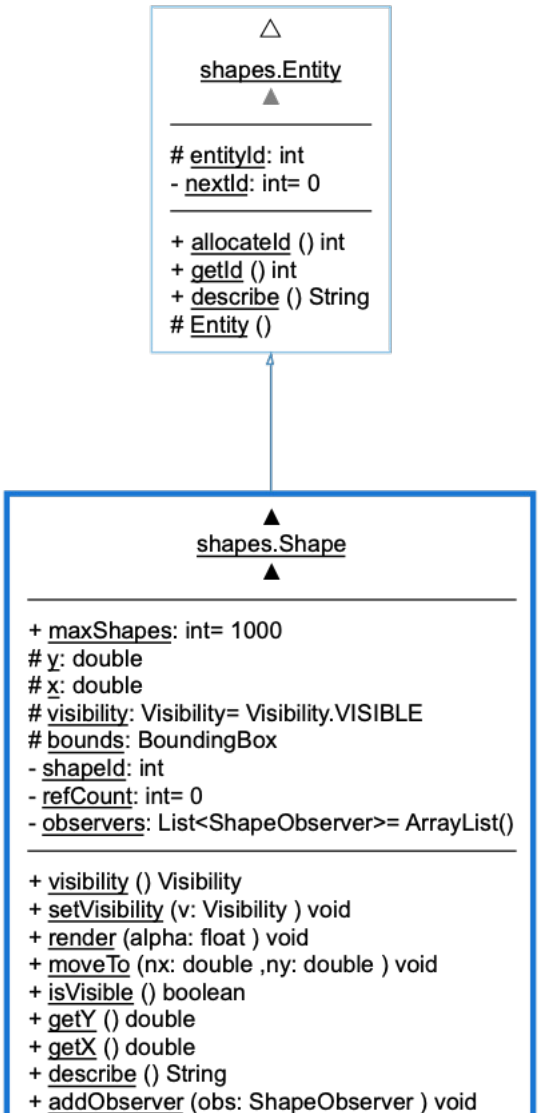
Rooted at the abstract class Shape below, the main box lists its private, protected, and public members in UML notation. Entity, its base class, is added above; Circle, its only subclass, is added below. Although Shape also implements Drawable and Renderable, neither interface appears, since this graph only follows class inheritance.

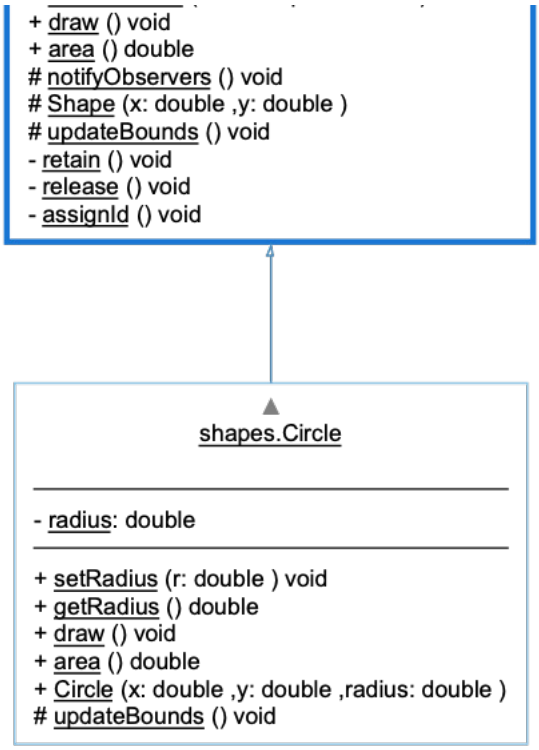
See the following code and corresponding graph

```
1 public abstract class Entity {
2     private static int nextId = 0;
3 
4     protected final int entityId;
5 
6     protected Entity() {
7         this.entityId = ++nextId;
8     }
9 
10     public    int    getId()      { return entityId; }
11     public abstract String describe();
12 
13     static int allocateId() { return ++nextId; }
14 }
15 
16 public interface Drawable {
17     void   draw();
18     double area();
19     default String drawDescription() { return "Drawable"; }
20 }
21 
22 interface Renderable {
23     void    render(float alpha);
24     boolean isVisible();
25 }
26 
27 public abstract class Shape
28     extends Entity implements Drawable, Renderable {
29 
30     public enum Visibility { VISIBLE, HIDDEN, CLIPPED }
31 
32     public interface ShapeObserver {
33         void onShapeChanged(Shape shape);
34     }
35 
36     public static int maxShapes = 1000;
37 
38     public abstract double area();
39     public abstract void   draw();
40 
41     public double  getX()          { return x; }
42     public double  getY()          { return y; }
43     public void moveTo(double nx, double ny) {
44         x = nx; y = ny; updateBounds(); }
```

```
45      public Visibility visibility() { return visibility; }
46      public void    setVisibility(Visibility v) { visibility = v; }
47      public void addObserver(ShapeObserver obs) { observers.add(obs); }
48
49      public boolean isVisible()
50          { return visibility == Visibility.VISIBLE; }
51      public void    render(float alpha) { if (isVisible()) draw(); }
52
53      public String describe() {
54          return getClass().getSimpleName() + "(" + x + "," + y + ")"; }
55
56      protected static class BoundingBox {
57          public double minX, minY, maxX, maxY;
58          BoundingBox(double x, double y, double w, double h) {
59              minX = x; minY = y; maxX = x + w; maxY = y + h;
60          }
61      }
62
63      protected double     x, y;
64      protected Visibility visibility = Visibility.VISIBLE;
65      protected BoundingBox bounds;
66
67      protected abstract void  updateBounds();
68      protected void           notifyObservers() {
69          for (ShapeObserver o : observers) o.onShapeChanged(this);
70      }
71
72      private static class IdCounter {
73          private static int count = 0;
74          static int next() { return ++count; }
75      }
76
77      private int          shapeId;
78      private int          refCount   = 0;
79      private List observers = new ArrayList();
80
81      private void assignId()   { shapeId = IdCounter.next(); }
82      private void retain()     { refCount++; }
83      private void release()    { if (--refCount < 0) refCount = 0; }
84
85      protected Shape(double x, double y) {
86          super();
87          this.x = x;
88          this.y = y;
89          assignId();
90      }
91  }
92
93  public class Circle extends Shape {
94
95      private double radius;
96
97      public Circle(double x, double y, double radius) {
```

```
98          super(x, y);
99          this.radius = radius;
100     }
101
102     public double  getRadius()     { return radius; }
103     public void    setRadius(double r) { radius = r; updateBounds(); }
104
105     @Override
106     public double area() { return Math.PI * radius * radius; }
107
108     @Override
109     public void    draw()          { }
110
111     @Override
112     protected void updateBounds()  {
113         bounds = new BoundingBox(
114             x - radius, y - radius, radius * 2, radius * 2);
115     }
116 }
```





UML Sequence Diagram

Languages: Ada, C#, C++, Java, Pascal, Python
Targets: Functions, Subprograms
Variants: Classic, Compare

Show a UML sequence diagram of the calls a selected function, method, procedure, or task makes, in the order they occur.

Each column is an actor — the class, file, or other scope a called entity belongs to — with a vertical lifeline running down it. A solid arrow is a call, landing on an activation box on the callee's lifeline; a dashed arrow returning to the caller is labeled with the callee's return type. An if, switch, or similar conditional is drawn as an alt combined fragment labeled with its condition, with each branch as its own section inside; a for, while, or do while loop is drawn the same way as a loop fragment labeled with its header. These fragments can nest to any depth, following the actual control flow of the function.

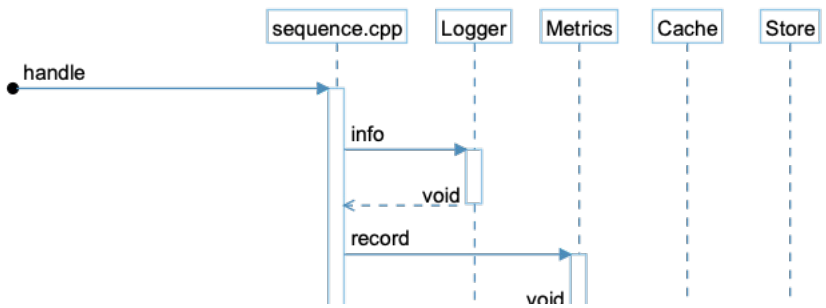
Rooted at handle below, the leftmost column is sequence.cpp (the file handle is defined in, since it isn't itself a member of a class), followed by a column for each struct it calls into: Logger, Metrics, Cache, and Store. The calls to log.info and metrics.record happen unconditionally; cache.has(id) is both the condition of the first alt fragment and, since evaluating it is itself a call, the first message shown inside it, followed by metrics.record and cache.get when it's true. store.fetch happens next, followed by a second alt fragment for val < 0 (log.error and

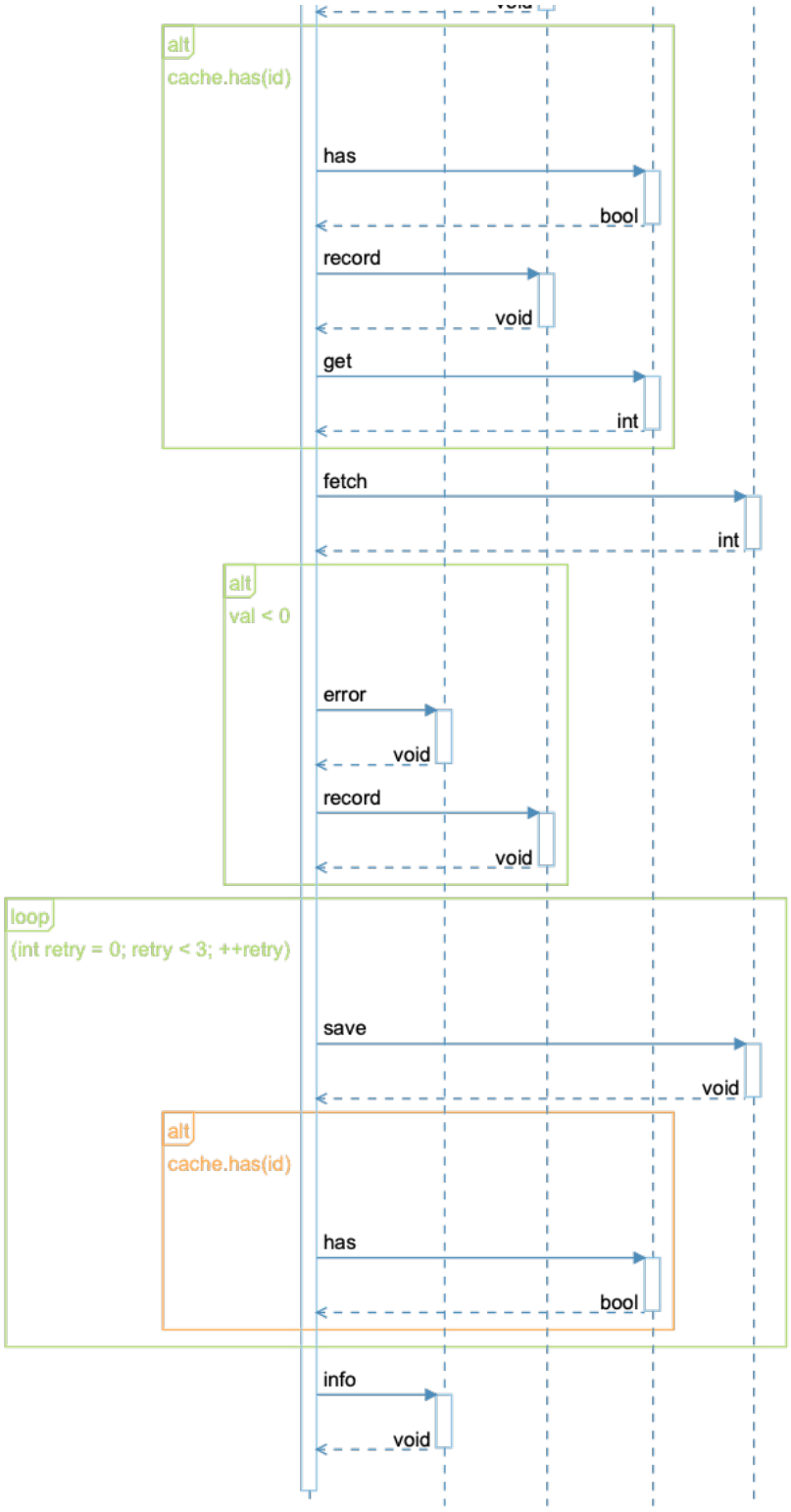
metrics.record), and a loop fragment for the retry loop, which calls store.save and checks cache.has(id) again in a nested alt before breaking. Finally, log.info is called once more after the loop.

See the following code and corresponding graph

```
1 struct Cache   { bool has(int id); int  get(int id); };
2 struct Store   { int  fetch(int id); void save(int id, int val); };
3 struct Logger {
4   void info(const char* msg);
5   void error(const char* msg);
6 };
7 struct Metrics { void record(const char* key, int val); };
8
9 int handle(int id, Cache& cache, Store& store,
10            Logger& log, Metrics& metrics) {
11     log.info("handling");
12     metrics.record("requests", 1);
13
14     if (cache.has(id)) {
15         metrics.record("hits", 1);
16         return cache.get(id);
17     }
18
19     int val = store.fetch(id);
20     if (val < 0) {
21         log.error("fetch failed");
22         metrics.record("errors", 1);
23         return -1;
24     }
25
26     for (int retry = 0; retry < 3; ++retry) {
27         store.save(id, val);
28         if (cache.has(id))
29             break;
30     }
31
32     log.info("done");
33     return val;
34 }
```

UML Sequence Diagram - Classic

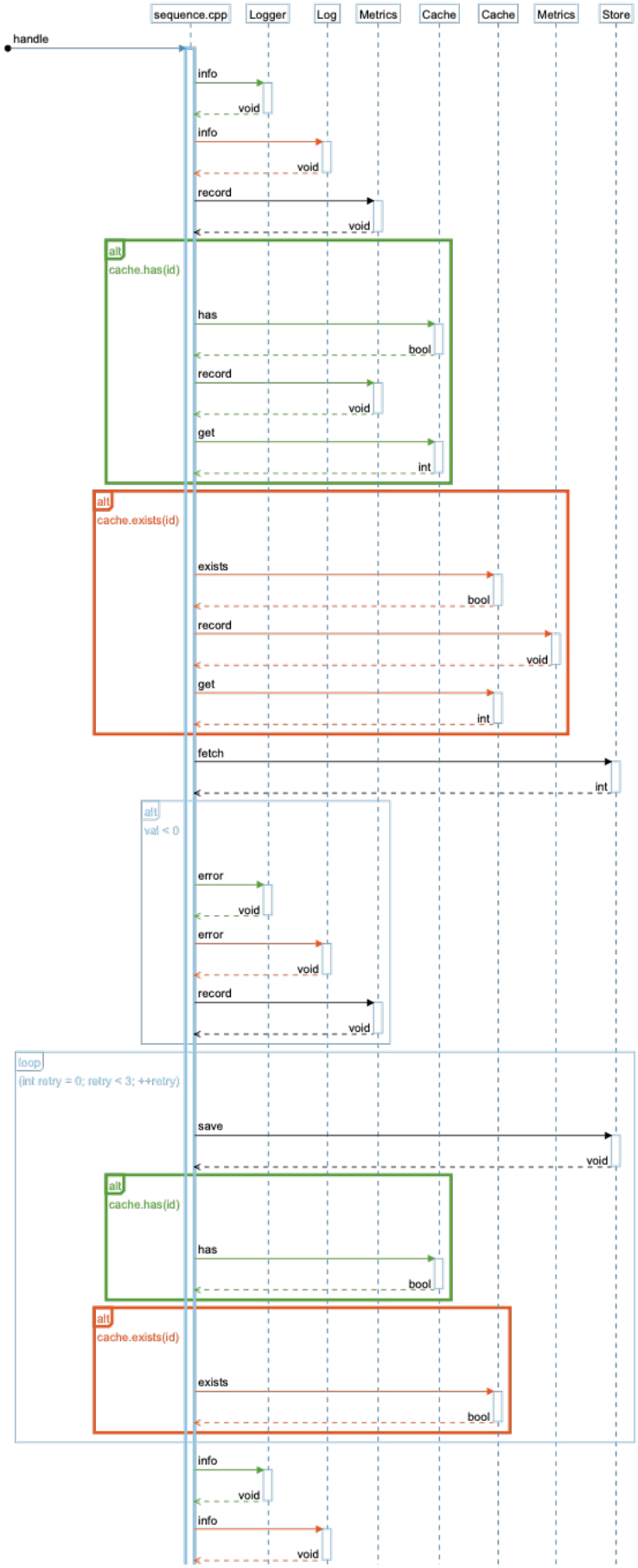




Back to UML Sequence Diagram

UML Sequence Diagram - Compare

The Compare variant shows how the graph has changed relative to the project's comparison database.


sequence.cpp
Logger
Log
Metrics
Cache
Cache
Metrics
Store
handle
info
void
info
void
record
void
alt
cache.has(id)
has
bool
record
void
get
int
alt
cache.exists(id)
exists
bool
record
void
get
int
fetch
int
alt
val < 0
error
void
error
void
record
void
loop
(int retry = 0; retry < 3; ++retry)
save
void
alt
cache.has(id)
has
bool
alt
cache.exists(id)
exists
bool
info
void
info
void



Back to UML Sequence Diagram

Used By

Languages: Fortran, VHDL, Web
Targets: Classes, Modules, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show what uses a selected Fortran module, VHDL design unit, or PHP trait, directly or indirectly.

For a selected Fortran module or VHDL architecture, configuration, entity, or package, this shows an edge from each module or design unit that uses it, then recurses into each of those units' own users, building a multi-level tree. For a selected PHP trait, this shows an edge from each class or trait that uses it in a use statement, then recurses the same way into whatever uses those in turn.

Rooted at math_constants below, the graph shows it used by vector_ops and matrix_ops, both of which are used by linear_algebra, which is itself used by statistics and solver. statistics and solver are each used by the program simulation, which nothing uses. For Fortran and VHDL, the inverse graph, Uses, shows the reverse relationship — everything the selected unit uses, directly or indirectly. PHP works the same way but with a different inverse graph: a trait like Auditable is used by Trackable, which is in turn used by ActiveRecord and Cacheable; the reverse of that relationship is shown by Uses Traits, not Uses.

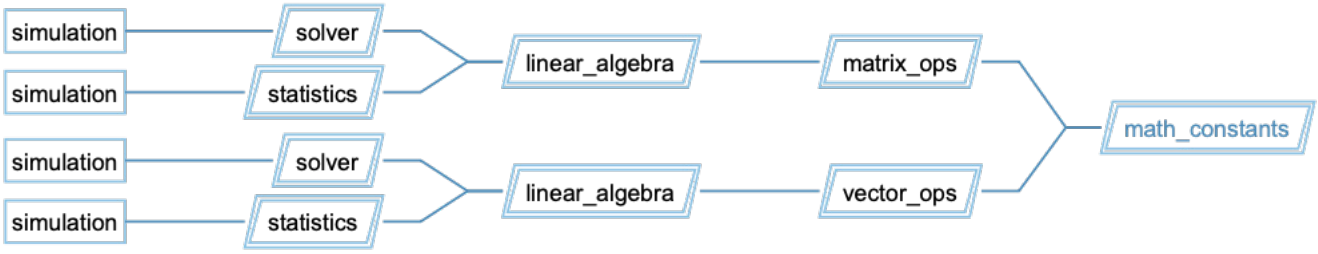
See the following code and corresponding graph

```
1 module math_constants
2 end module math_constants
3
4 module vector_ops
5   use math_constants
6 end module vector_ops
7
8 module matrix_ops
9   use math_constants
10 end module matrix_ops
11
12 module linear_algebra
13   use vector_ops
14   use matrix_ops
15 end module linear_algebra
16
17 module statistics
18   use linear_algebra
```

```
19 end module statistics
20
21 module solver
22   use linear_algebra
23 end module solver
24
25 program simulation
26   use solver
27   use statistics
28 end program simulation
```

Used By - Classic

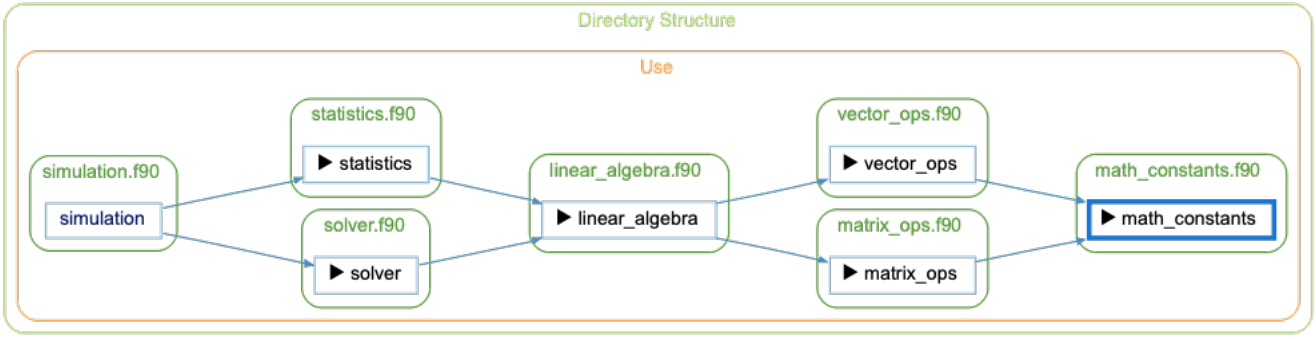
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Used By

Used By - Cluster

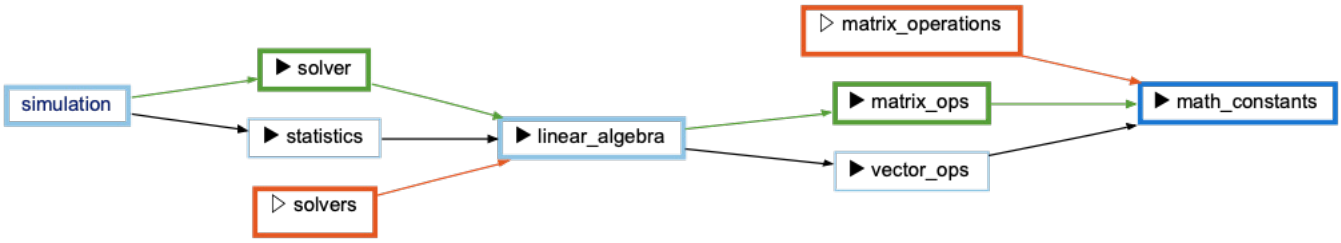
The Cluster variant groups entities by their containing class/file/architecture.



Back to Used By

Used By - Compare

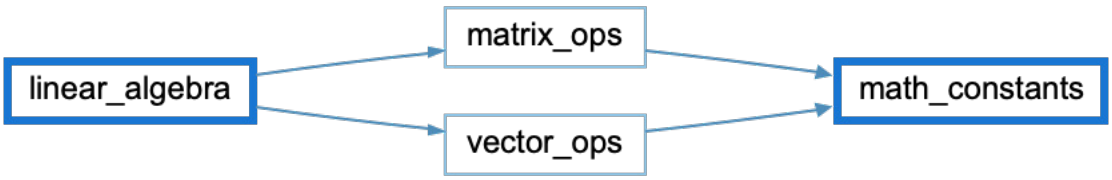
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Used By

Used By - Relationship

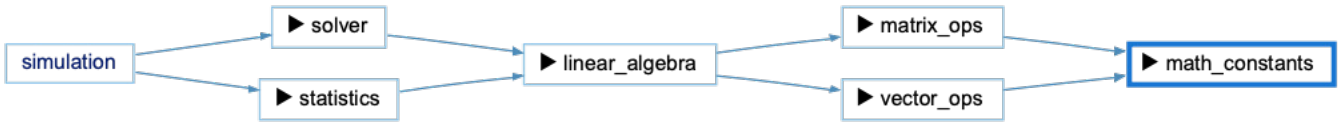
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Used By

Used By - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Used By

Uses

Languages: Fortran, VHDL
Targets: Modules, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the Fortran modules or VHDL design units a selected module or design unit uses, directly or indirectly.

For a selected Fortran module, block data, program, function, subroutine, or procedure, this shows an edge to each module it uses, then recurses into each of those modules' own uses, building a multi-level tree. For a selected VHDL architecture, configuration, entity, or package, this shows the same relationship for

VHDL's use clause.

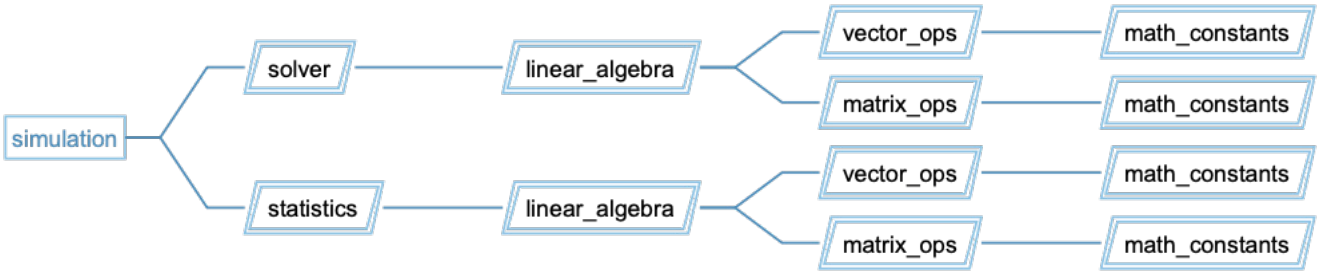
Rooted at the program simulation below, the graph shows it using solver and statistics, both of which use linear_algebra. linear_algebra itself uses vector_ops and matrix_ops, and each of those uses math_constants, which uses nothing further. The inverse graph, Used By, shows the reverse relationship — everything that uses the selected module or design unit, directly or indirectly.

See the following code and corresponding graph

```
1 module math_constants
2 end module math_constants
3
4 module vector_ops
5   use math_constants
6 end module vector_ops
7
8 module matrix_ops
9   use math_constants
10 end module matrix_ops
11
12 module linear_algebra
13   use vector_ops
14   use matrix_ops
15 end module linear_algebra
16
17 module statistics
18   use linear_algebra
19 end module statistics
20
21 module solver
22   use linear_algebra
23 end module solver
24
25 program simulation
26   use solver
27   use statistics
28 end program simulation
```

Uses - Classic

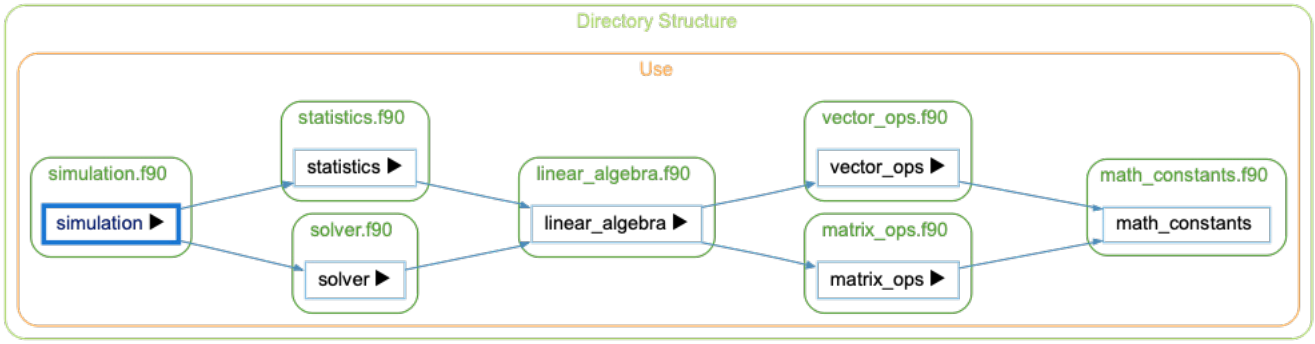
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Uses

Uses - Cluster

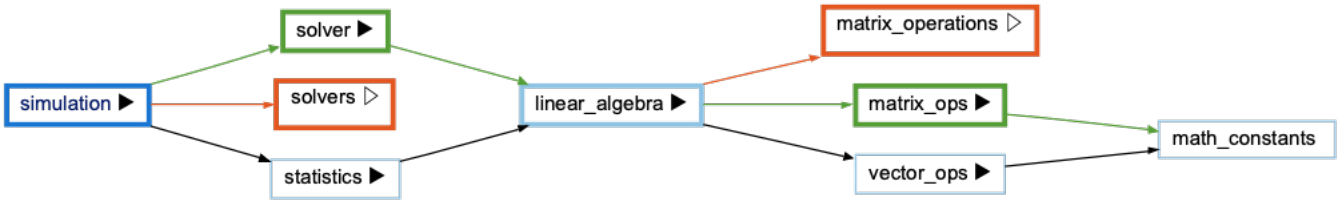
The Cluster variant groups entities by their containing class/file/architecture.



Back to Uses

Uses - Compare

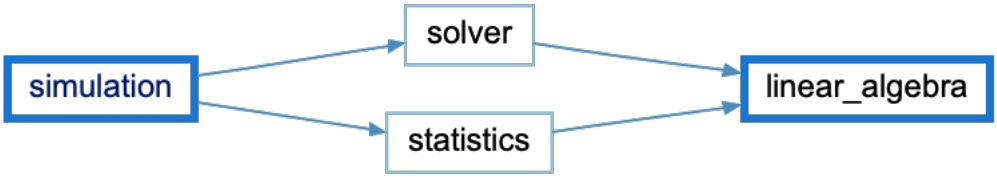
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Uses

Uses - Relationship

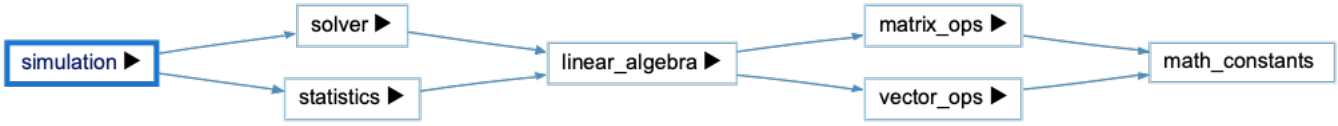
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Uses

Uses - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Uses

Uses & Used By

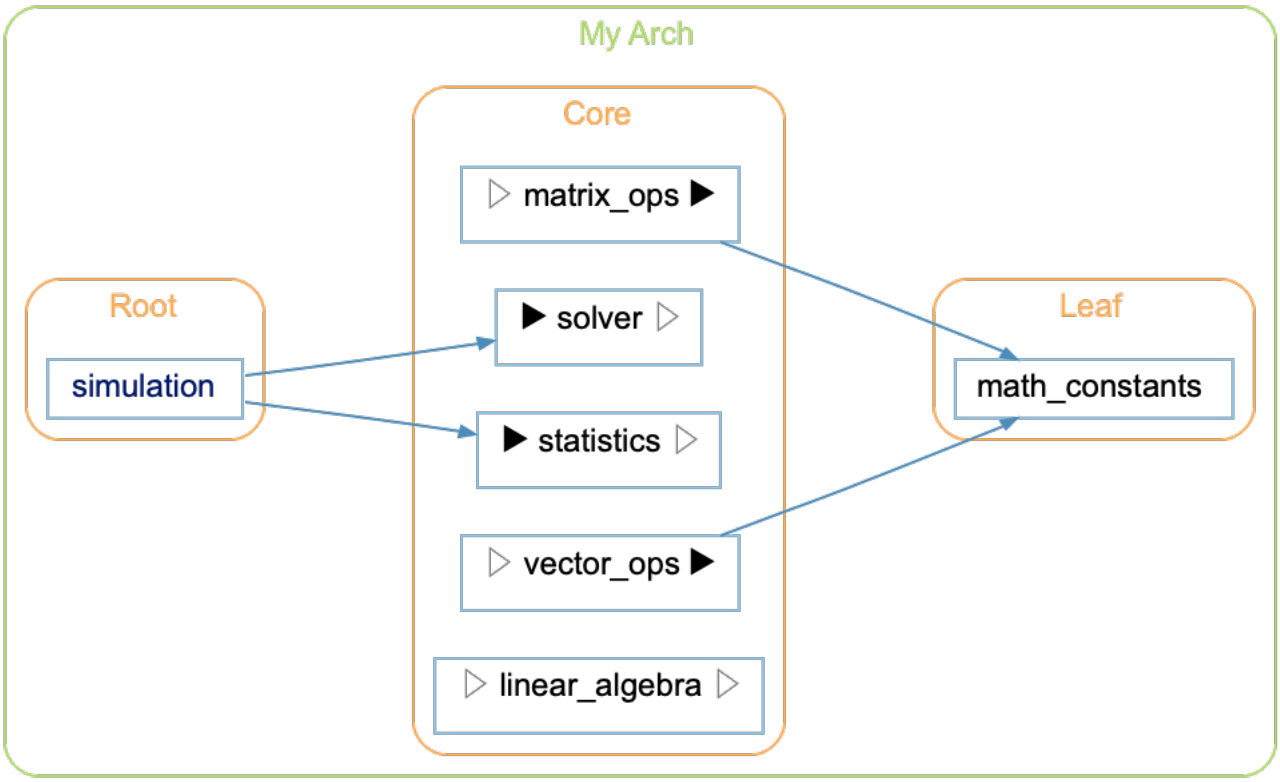
Languages: Fortran, VHDL
Targets: Architectures
Variants: Butterfly, Internal, Used By, Uses

Show Uses and Used By relationships for an architecture

See Uses and Used By graphs for details

Uses & Used By - Butterfly

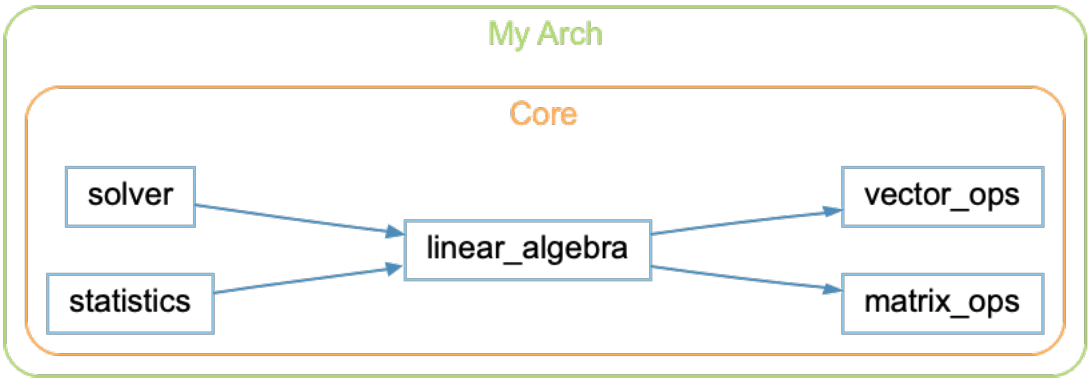
Show both Uses relationships from and Used By relationships to entities in an architecture



Back to Uses & Used By

Uses & Used By - Internal

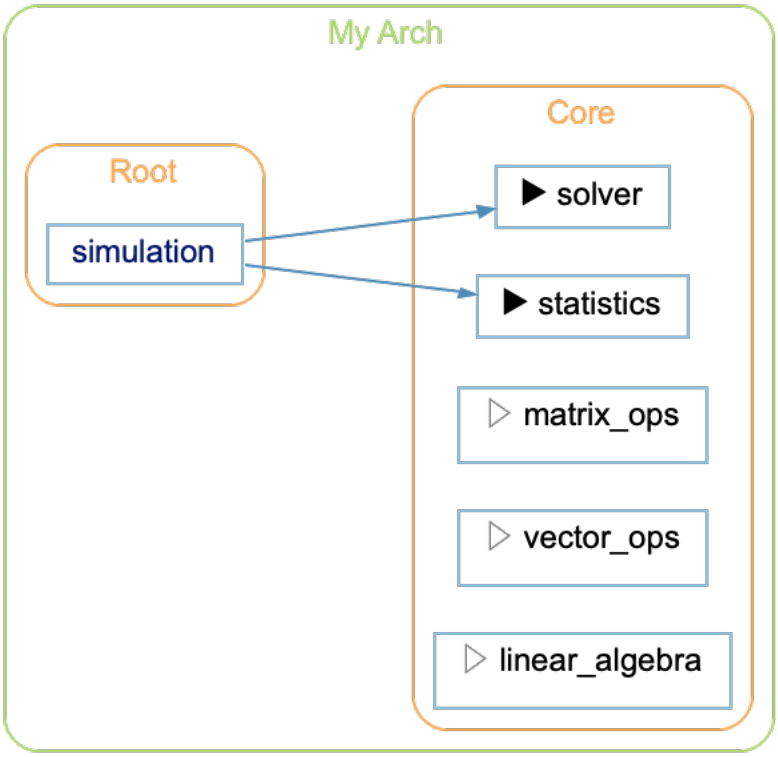
Show Uses relationships between entities in an architecture



Back to Uses & Used By

Uses & Used By - Used By

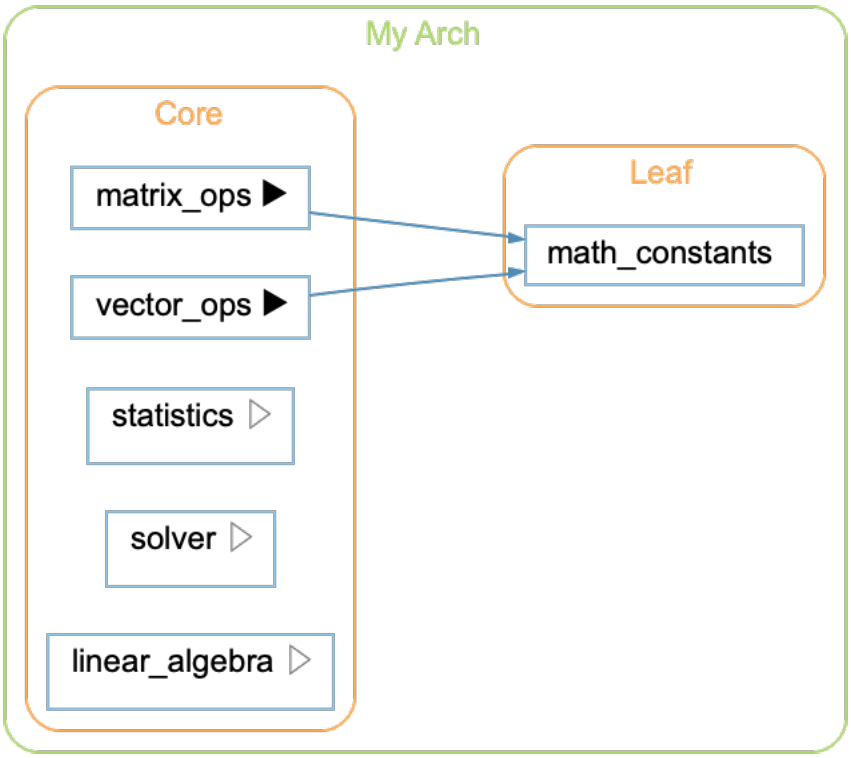
Show Used By relationships to entities in an architecture



Back to Uses & Used By

Uses & Used By - Uses

Show Uses relationships from entities in an architecture



Back to Uses & Used By

Uses Traits

Languages: Web
Targets: Classes
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the PHP traits a selected class or trait uses, directly or indirectly.

For a selected PHP class or trait, this shows an edge to each trait it uses in a use statement, then recurses into each of those traits' own uses, building a multi-level tree.

Rooted at Trackable below, the graph shows it using Auditable and Sluggable, which use Timestampable and Identifiable respectively, neither of which uses anything further. The inverse graph, Used By, shows the reverse relationship — everything that uses the selected class or trait, directly or indirectly.

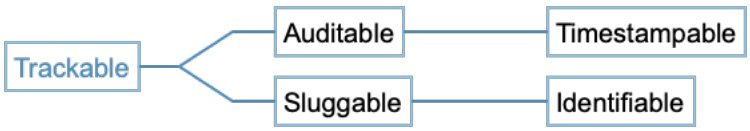
See the following code and corresponding graph

```
trait Trackable {
    use Auditable;
```

```
    use Sluggable;
}

trait Auditable {
    use Timestampable;
}

trait Sluggable {
    use Identifiable;
}

trait Timestampable {
}

trait Identifiable {
}
```

Uses Traits - Classic

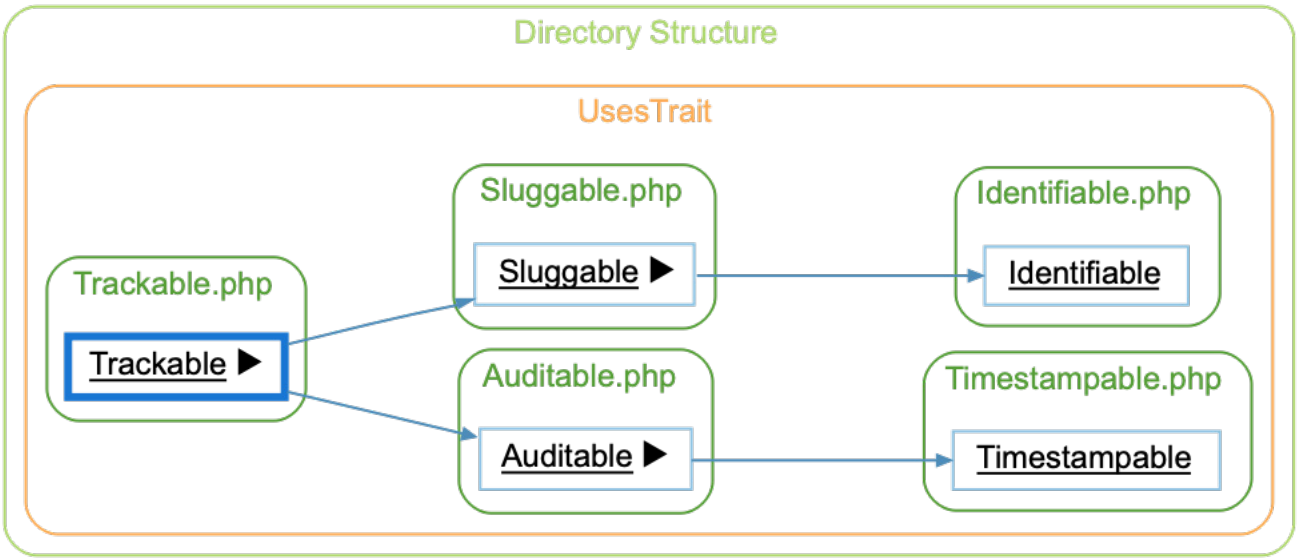
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Uses Traits

Uses Traits - Cluster

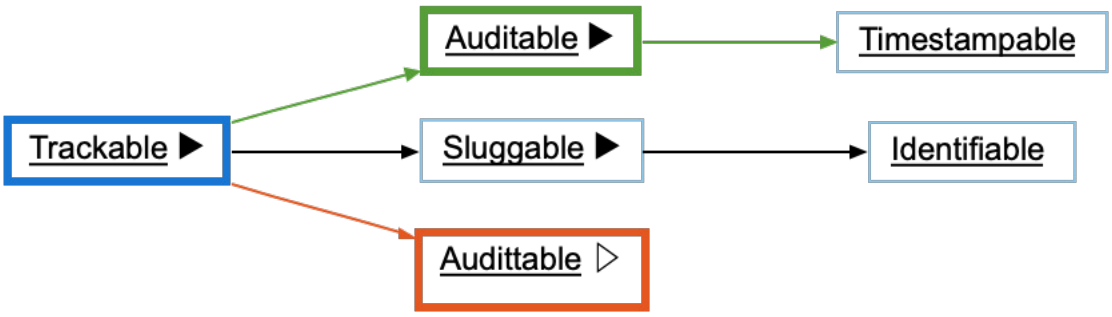
The Cluster variant groups entities by their containing class/file/architecture.



Back to Uses Traits

Uses Traits - Compare

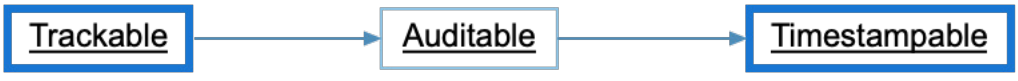
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Uses Traits

Uses Traits - Relationship

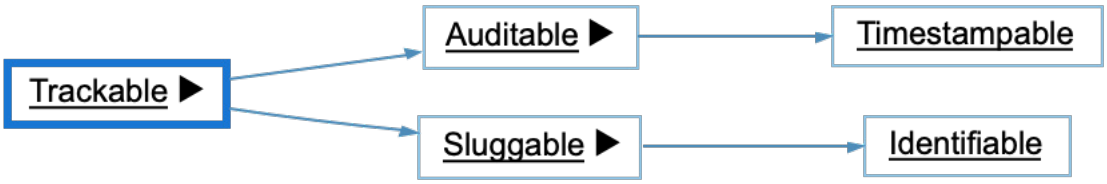
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Uses Traits

Uses Traits - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Uses Traits

Uses Traits & Used By

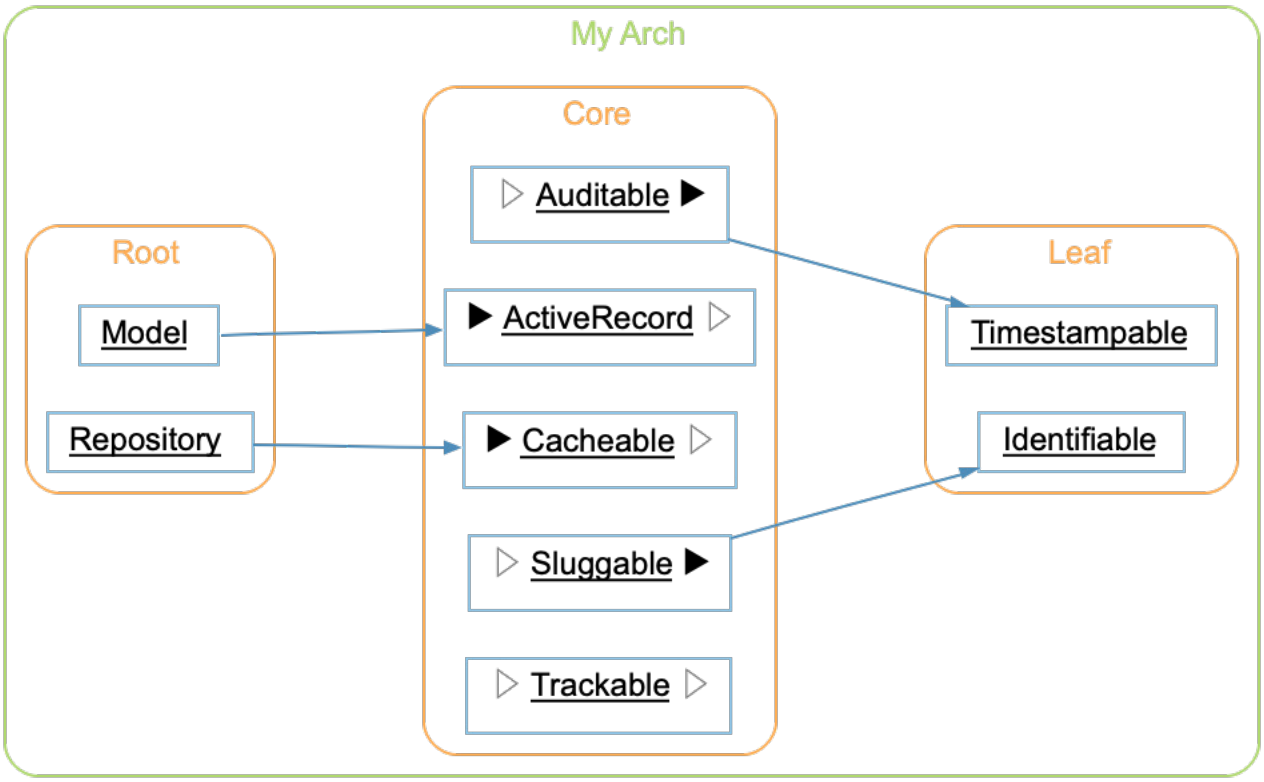
Languages: Web
Targets: Architectures
Variants: Butterfly, Internal, Used By, Uses Traits

Show Uses Traits and Used By relationships for an architecture

See Uses Traits and Used By graphs for details

Uses Traits & Used By - Butterfly

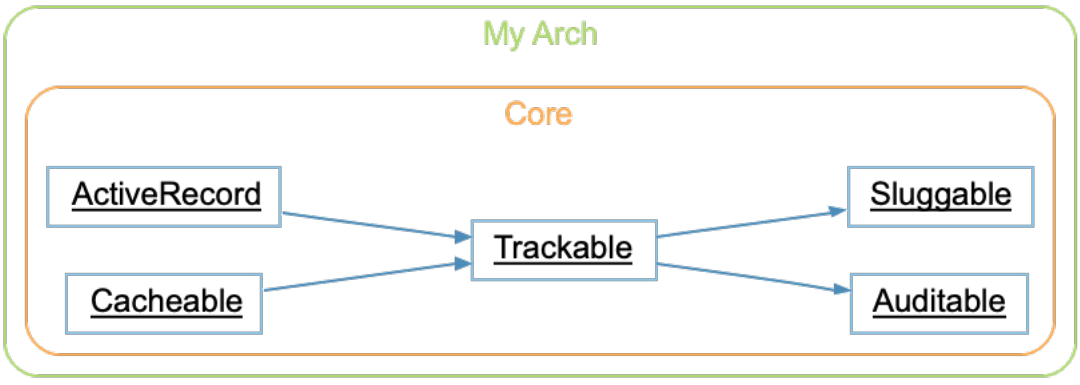
Show both Uses Traits relationships from and Used By relationships to entities in an architecture



Back to Uses Traits & Used By

Uses Traits & Used By - Internal

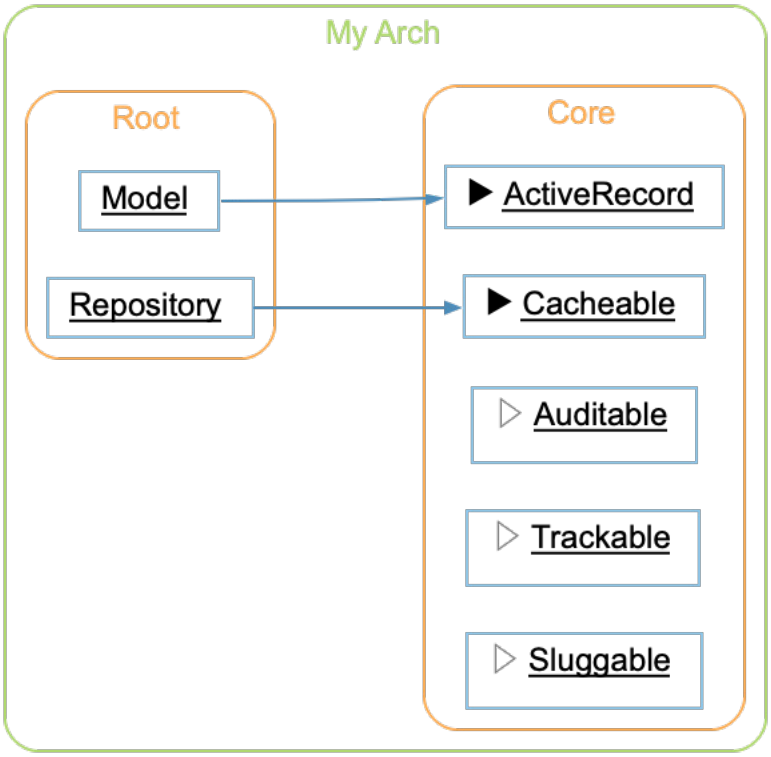
Show Uses Traits relationships between entities in an architecture



Back to Uses Traits & Used By

Uses Traits & Used By - Used By

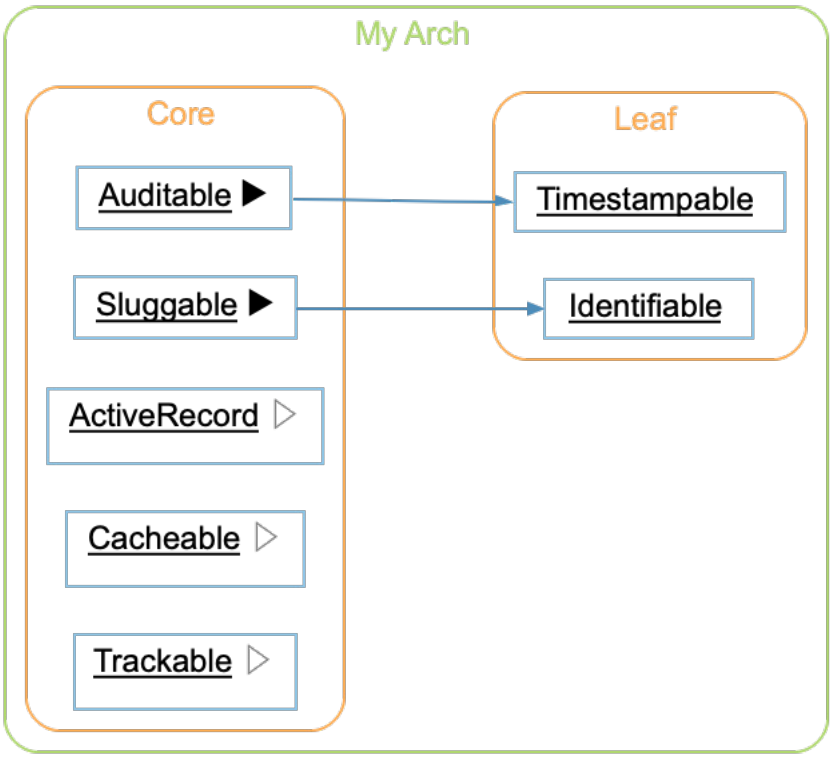
Show Used By relationships to entities in an architecture



Back to Uses Traits & Used By

Uses Traits & Used By - Uses Traits

Show Uses Traits relationships from entities in an architecture



Back to Uses Traits & Used By

Variable Tracker

Languages: C#, C++, Java
Targets: Objects
Variants: Classic

Show the source code of everything a selected object, variable, or parameter's value is directly or indirectly built from.

For a selected object, variable, or parameter, this shows the source code of each place that sets its value — an assignment, an increment or decrement, or an argument passed to a function's parameter — then recurses into every entity that appears in that code, building a multi-level tree. When a value comes from a function call, this continues into the called function's own return statements; when a value comes from a parameter, this continues into the argument passed at each call site.

See the Assignments graph for a related view of the same relationships. Assignments uses "assign" references and shows just the entities involved, so it skips over literals. Variable Tracker instead shows the actual source code at each step, following every entity that appears in it. That leads to a slight difference in code like x = func(y). Both graphs connect x to func(), using the function's return

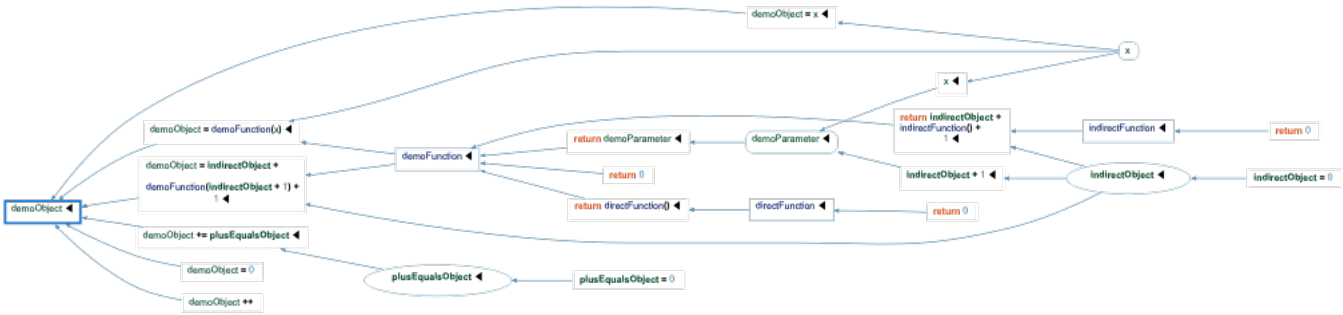
statements, but the Variable Tracker will additionally connect y because it's part of the same statement. Both graphs can follow assignments to a parameter from each call site.

Rooted at demoObject in myFunction below, the graph shows its direct inputs: x, plusEqualsObject, and the literal-only assignments demoObject = 0 and demoObject++ (shown with no further inputs of their own), plus its two calls to demoFunction. Since both calls reach the same demoFunction entity, it's shown once, expanding into each of its own return statements — including return demoParameter, which in turn is fed by whatever was passed at each call site, x and indirectObject + 1.

See the following code and corresponding graph

```
1 extern int directFunction();
2 extern int indirectFunction();
3 extern int indirectObject;
4 extern int plusEqualsObject;
5
6 int demoFunction(int demoParameter)
7 {
8   switch (demoParameter) {
9     case 0: return demoParameter; // demoParameter -> demoFunction()
10     case 1: return directFunction(); // -> demoFunction()
11     case 2: return 0; // literal, no reference
12     default:
13       // All indirect: part of an expression
14       return indirectObject + // indirectObject -> demoFunction()
15              indirectFunction() + // -> demoFunction()
16              1; // literal, no reference
17   }
18 }
19
20 void myFunction(int x)
21 {
22   // demoFunction() -> demoObject; x -> demoParameter
23   int demoObject = demoFunction(x);
24
25   demoObject = x; // x -> demoObject
26   demoObject = 0; // literal, no reference
27
28   // All indirect: part of an expression
29   demoObject = indirectObject + // indirectObject -> demoObject
30                // demoFunction() -> demoObject; x -> demoParameter
31                demoFunction(indirectObject + 1) +
32                1; //  literal, no reference
33
34   demoObject ++; // modification by literal, no reference
35
36   // modification by plusEqualsObject, indirect assignment
37   demoObject += plusEqualsObject;
```

```
38 }
```



With

Languages: Ada
Targets: Packages, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show the Ada packages a selected package, subprogram, task, or protected object withs, directly or indirectly.

For a selected Ada unit, this shows an edge to each package named in one of its with clauses — whether the clause appears on the unit's spec or its body — then recurses into each of those packages' own with clauses, building a multi-level tree.

Rooted at Application below, the graph shows edges to Pipeline and Monitor (withed in Application's spec) and to Processor (withed separately in Application's body). Processor in turn withs Data_Reader and Data_Writer (from its spec) and IO_Utils (from its body); Pipeline and Monitor each with Processor too, continuing the same pattern. IO_Utils withs nothing further, ending the chain. The inverse graph, Withby, shows the reverse relationship — everything that withs the selected unit, directly or indirectly.

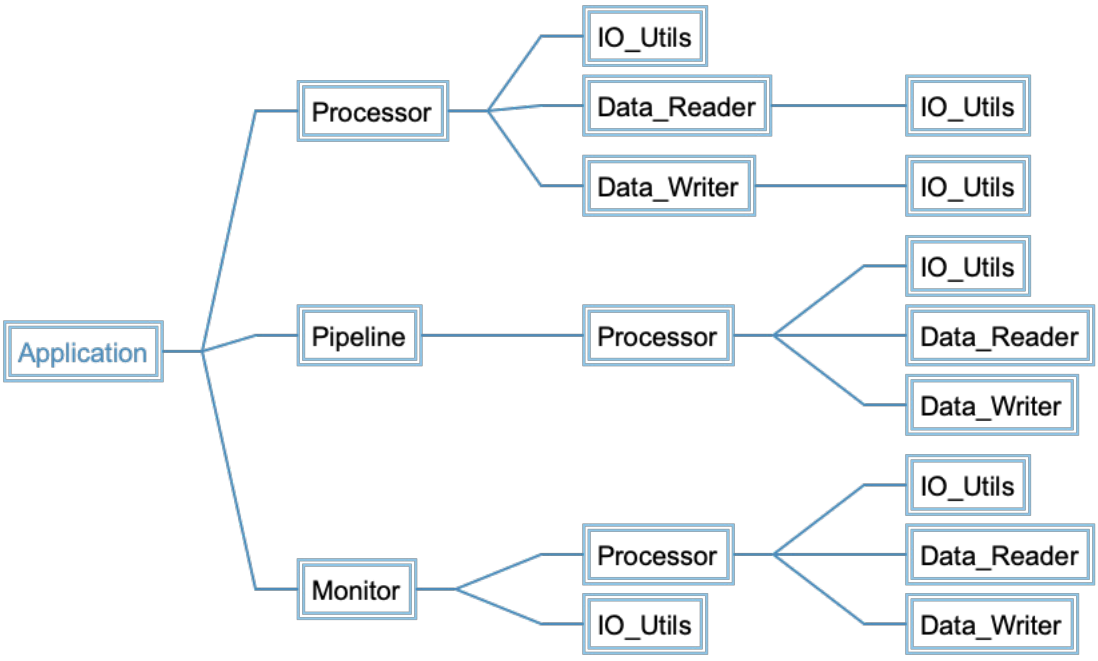
See the following code and corresponding graph

```
1 // application.ads
2 with Pipeline;
3 with Monitor;
4
5 // application.adb
6 with Processor;
7
8 // pipeline.ads
9 with Processor;
10
11 // monitor.ads
12 with Processor;
```

```
13
14 // monitor.adb
15 with IO_Utils;
16
17 // processor.ads
18 with Data_Reader;
19 with Data_Writer;
20
21 // processor.adb
22 with IO_Utils;
23
24 // data_reader.ads
25 with IO_Utils;
26
27 // data_writer.ads
28 with IO_Utils;
29
30 // io_utils.ads
```

With - Classic

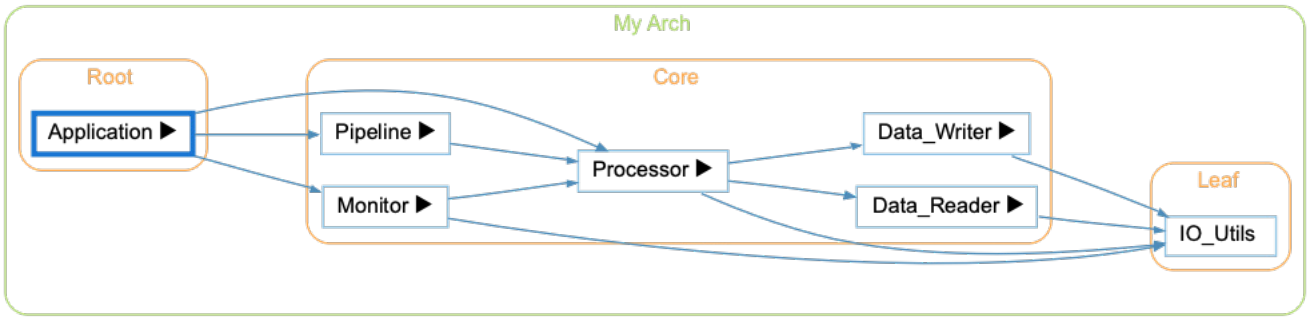
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to With

With - Cluster

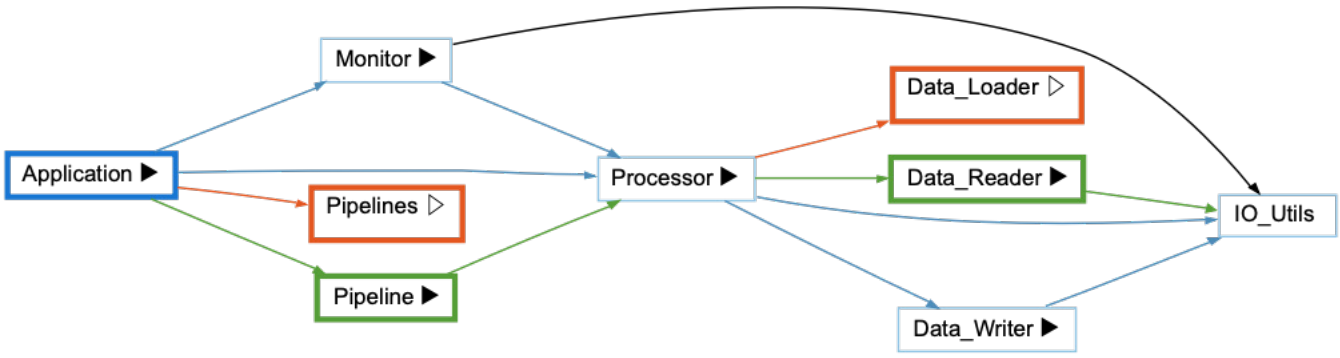
The Cluster variant groups entities by their containing class/file/architecture.



Back to With

With - Compare

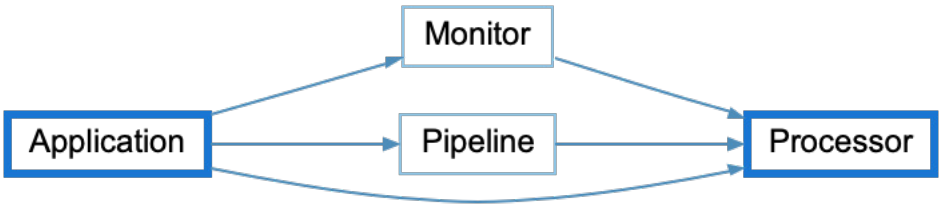
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to With

With - Relationship

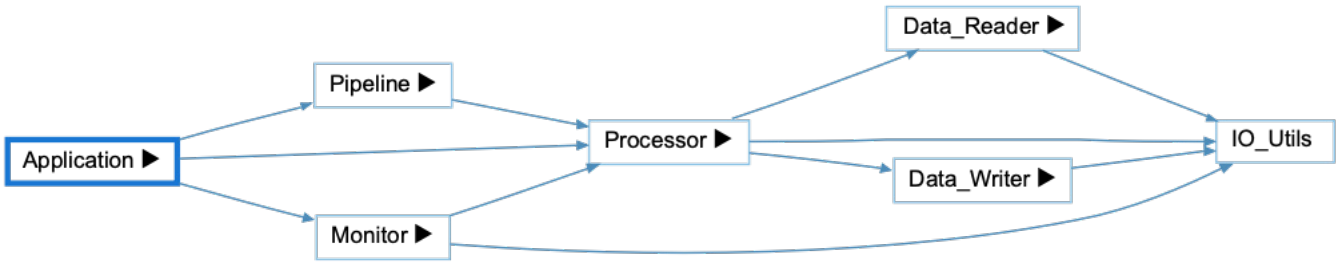
The Relationship variant filters the graph to only paths connecting the two entities.



Back to With

With - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to With

With & Withby

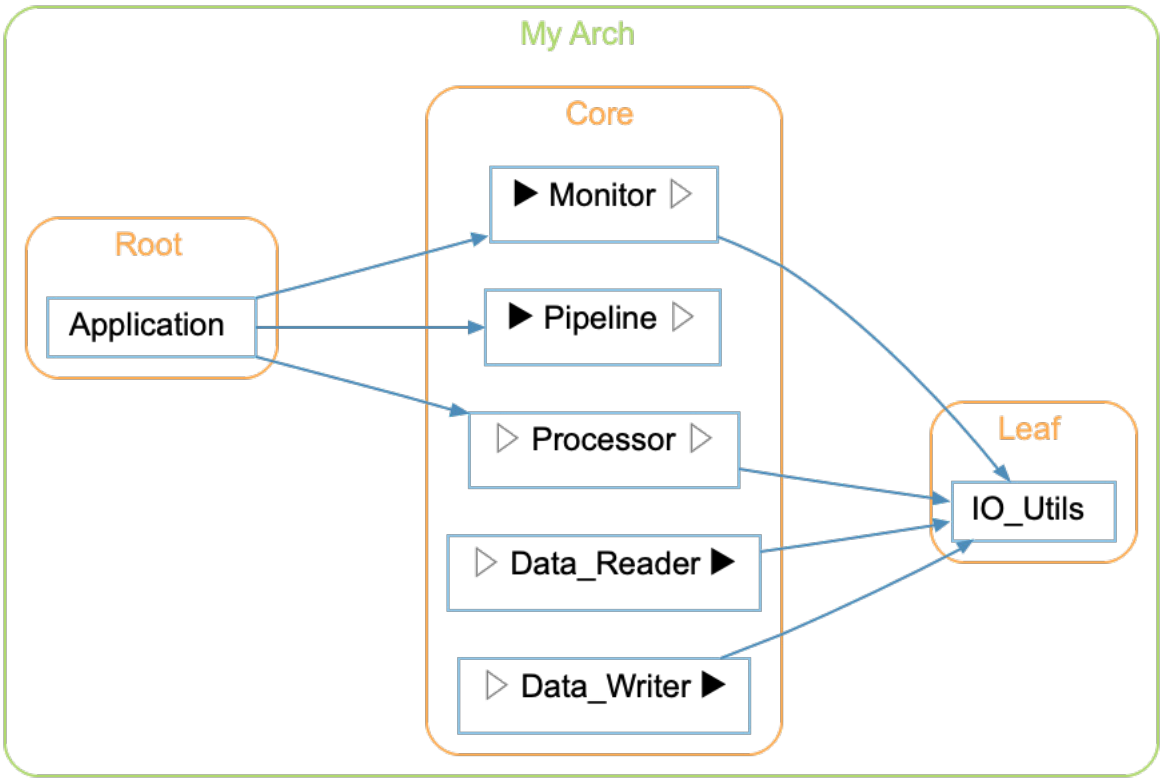
Languages: Ada
Targets: Architectures, Files
Variants: Butterfly, Internal, With, Withby

Show With and Withby relationships for a file or architecture

See With and Withby graphs for details

With & Withby - Butterfly

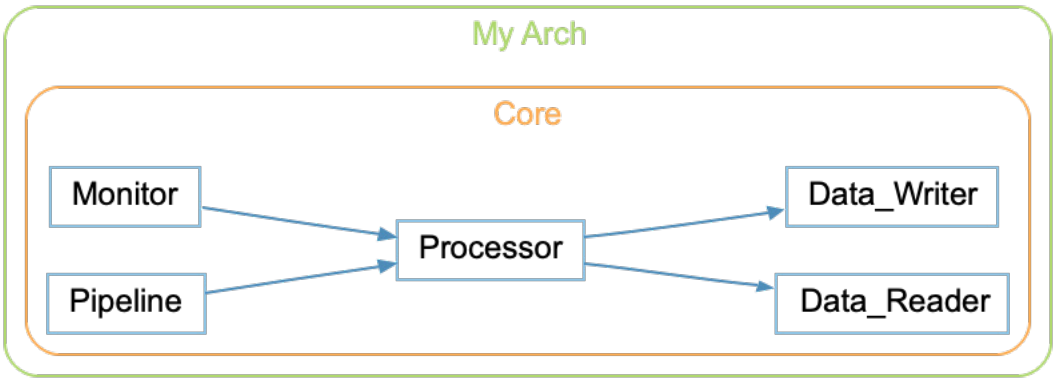
Show both With relationships from and Withby relationships to entities in a file or architecture



Back to With & Withby

With & Withby - Internal

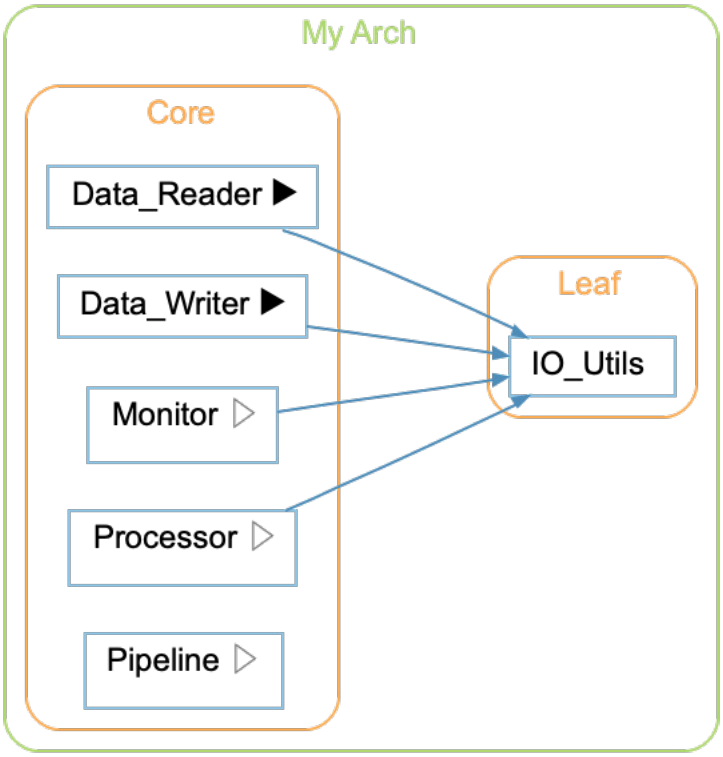
Show With relationships between entities in a file or architecture



Back to With & Withby

With & Withby - With

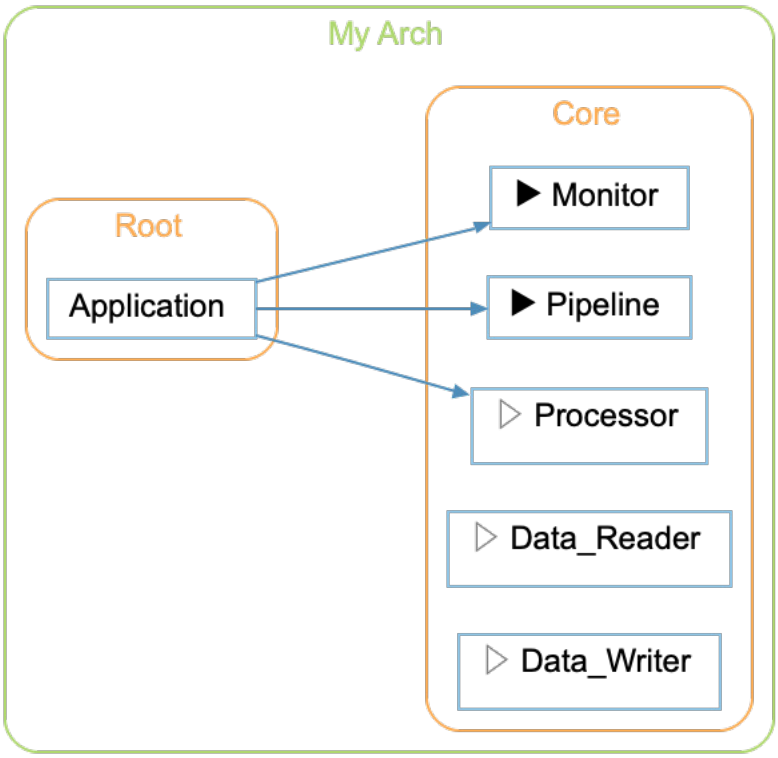
Show With relationships from entities in a file or architecture



Back to With & Withby

With & Withby - Withby

Show Withby relationships to entities in a file or architecture



Back to With & Withby

Withby

Languages: Ada
Targets: Packages, Subprograms
Variants: Classic, Cluster, Compare, Relationship, Simplified

Show what withs a selected Ada package, subprogram, task, or protected object, directly or indirectly.

For a selected Ada unit, this shows an edge from each package that names it in a with clause — whether that clause is on the withing package's spec or its body — then recurses into each of those packages' own withers, building a multi-level tree.

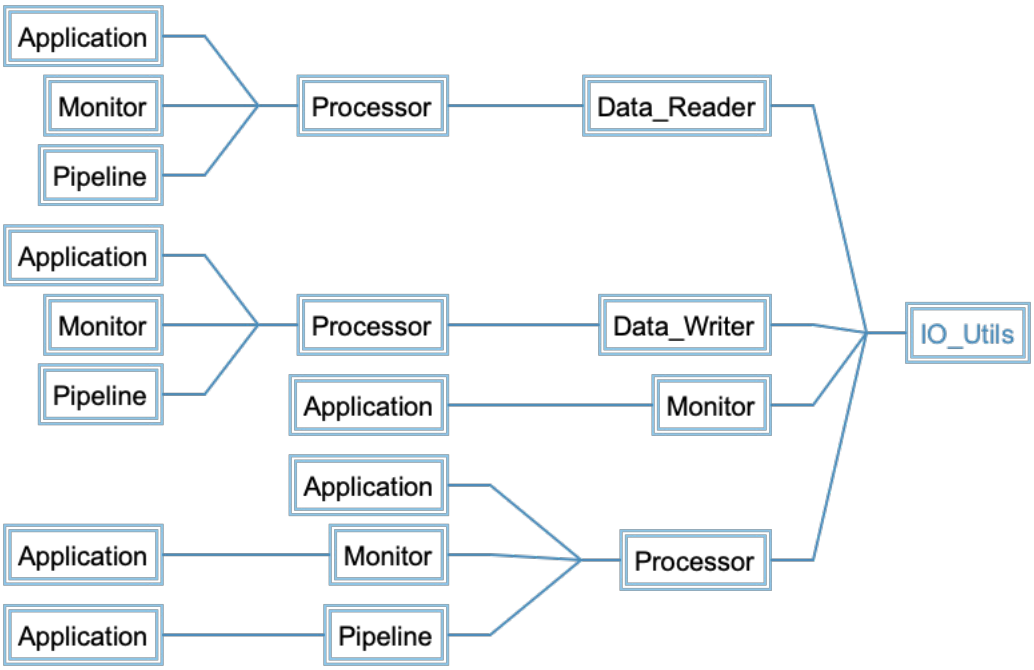
Rooted at IO_Utils below, the graph shows it withed by Data_Reader and Data_Writer (from their specs) and by Processor and Monitor (from their bodies). Processor is itself withed by Pipeline and Monitor (from their specs) and by Application (from its body); Pipeline and Monitor are each withed by Application's spec in turn. Application is withed by nothing, ending the chain. The inverse graph, With, shows the reverse relationship — everything the selected unit withs, directly or indirectly.

See the following code and corresponding graph

```
1 // application.ads
2 with Pipeline;
3 with Monitor;
4
5 // application.adb
6 with Processor;
7
8 // pipeline.ads
9 with Processor;
10
11 // monitor.ads
12 with Processor;
13
14 // monitor.adb
15 with IO_Utils;
16
17 // processor.ads
18 with Data_Reader;
19 with Data_Writer;
20
21 // processor.adb
22 with IO_Utils;
23
24 // data_reader.ads
25 with IO_Utils;
26
27 // data_writer.ads
28 with IO_Utils;
29
30 // io_utils.ads
```

Withby - Classic

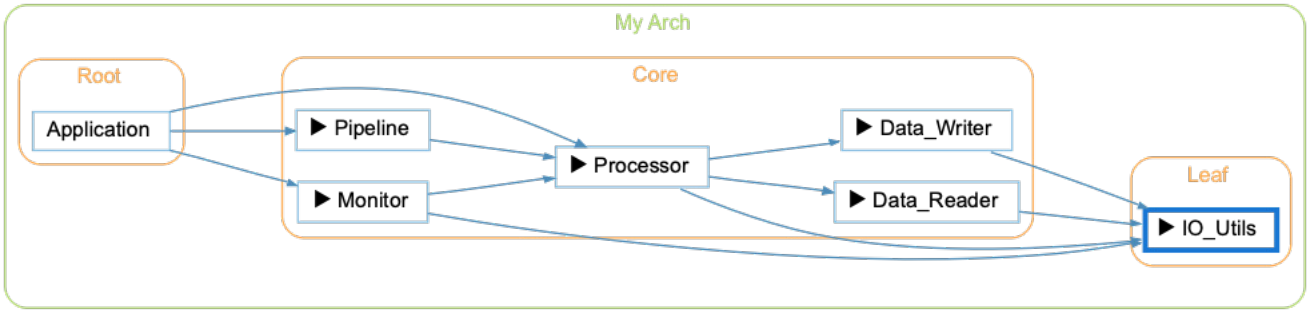
The Classic variant is similar to the Simplified variant, showing all relationships in a simple tree, but it uses a custom layout algorithm instead of Graphviz.



Back to Withby

Withby - Cluster

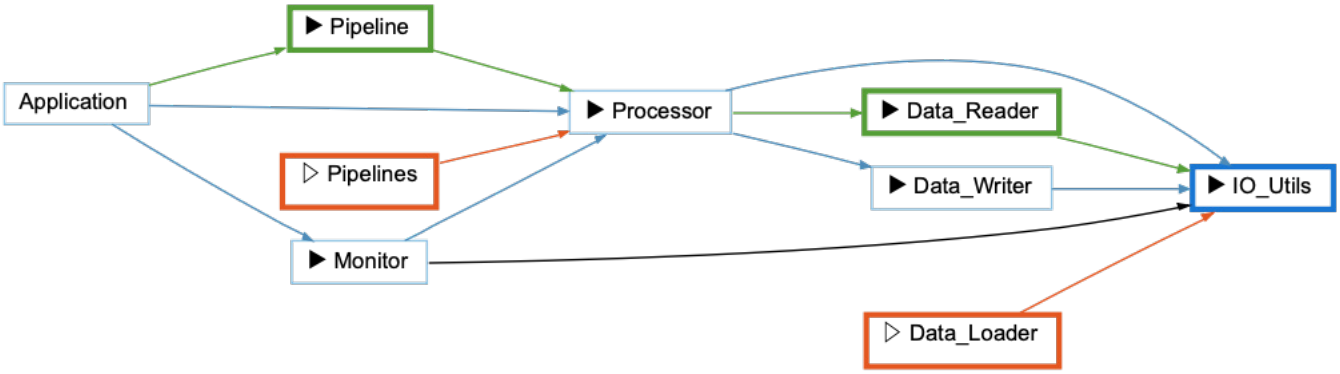
The Cluster variant groups entities by their containing class/file/architecture.



Back to Withby

Withby - Compare

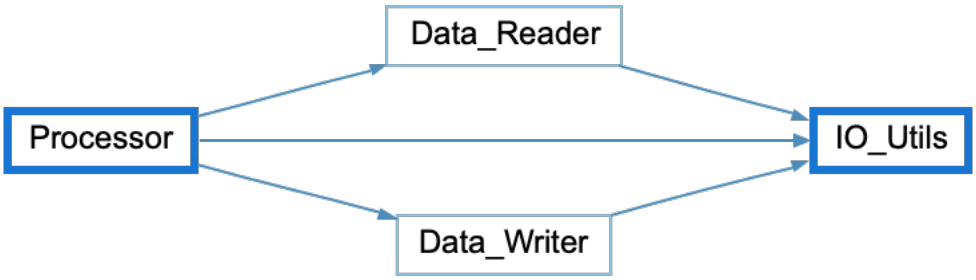
The Compare variant shows how the graph has changed relative to the project's comparison database.



Back to Withby

Withby - Relationship

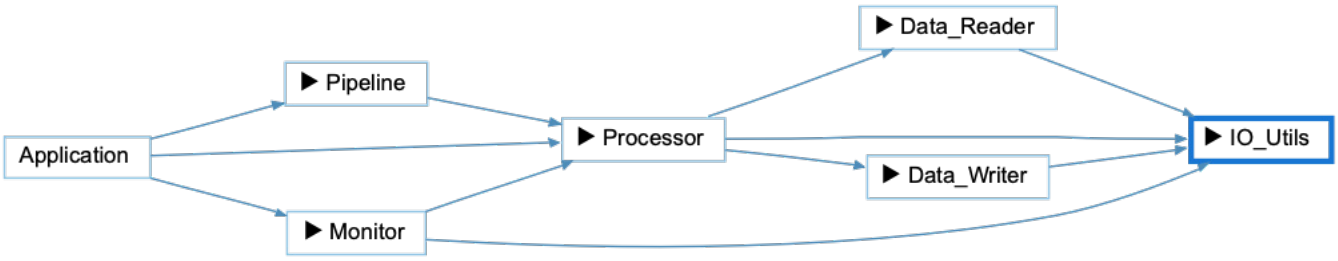
The Relationship variant filters the graph to only paths connecting the two entities.



Back to Withby

Withby - Simplified

The Simplified variant shows all relationships in a simple tree (no clusters)



Back to Withby