== FILE: docs/classCoupling.doc.md

Class Coupling Metric

ID

`class-coupling`

Name

Class Coupling

Description

Analyzes each class to identify **fan-out** (methods called from other classes) and **fan-in** (methods called by other classes).
This metric provides a **method-level dependency map** between classes to understand coupling in the codebase.

- **Fan-Out**: methods in the current class that call methods of other classes.
- **Fan-In**: methods in the current class that are called by methods of other classes.

Output Format (example)

```json
{
  "name": "Class Coupling",
  "description": "Analyzes each class to identify Fan-Out and Fan-In",
  "result": {
    "/path/to/DeclaredClass.ts": {
      "DeclaredClass": [
        {
          "type": "ClassMethod",
          "key": { "type": "Identifier", "name": "declaredMethod" },
          "params": [],
          "body": { "type": "BlockStatement", "body": [] },
          "fan-in": {
            "ExpressedClass": { "expressedMethod": 2 }
          }
        }
      ]
    },
    "/path/to/ExpressedClass.ts": {
      "ExpressedClass": [
        {
          "type": "ClassMethod",
          "key": { "type": "Identifier", "name": "constructor" },
          "params": [],
          "body": { "type": "BlockStatement", "body": [] }
        },
        {
          "type": "ClassMethod",
          "key": { "type": "Identifier", "name": "expressedMethod" },
          "params": [],
          "body": { "type": "BlockStatement", "body": [] },
          "fan-out": {
            "DeclaredClass": { "declaredMethod": 2 }
          }
        }
      ]
    }
  },
  "status": true
}
```

How it works

1. **Dependencies**
    - Requires `classes-per-file` to know all classes and their methods.
    - Requires `instance-mapper` to resolve which instances belong to which classes.
2. **Traversal**
    - Detects **ClassExpression** assigned to variables.
    - Traverses each class for:
        - **ClassMethod**
        - **ClassProperty** containing `ArrowFunctionExpression` or `FunctionExpression`.
3. **Fan-Out Recording**
    - For each **NewExpression** (e.g., `new ClassF()`), maps to the constructor (`_constructor`) of the instantiated class.
    - For each **CallExpression**:
        - Resolves method calls on instances and direct class calls.
        - Handles `this.property.method()` patterns using `instance-mapper`.
    - Increments `fan-out` counts for the caller method toward the callee method.
4. **Fan-In Recording**
    - Automatically updates `fan-in` for callee methods based on the caller's fan-out.

5. **Post-processing**

    - Cleans temporary fields (`currentFile`, `dependencies`) and sets `status` to true.

Notes

- Tracks **method-level dependencies**, not just class-level.
- Handles both **methods** and **function properties** inside classes.
- `_constructor` is used internally to avoid collisions with JavaScript's reserved `constructor`.
- Useful to identify:
    - Highly coupled classes
    - Potential refactoring opportunities
    - Method-level dependencies for architectural analysis

Use cases

- Measure **class coupling** to assess maintainability and modularity.
- Detect **fan-in hotspots** (methods heavily used by others) and **fan-out responsibilities** (methods calling many external methods).
- Serves as input for higher-level analysis, such as **system complexity** or **dependency graphs**.

==

FILE: docs/classesPerFile.doc.md

Classes Per File Metric

ID

`classes-per-file`

Name

Classes Per File

Description

Analyzes each source file to identify and record all **top-level classes** defined.
For each file, it stores the classes found as keys and lists their **methods** and **properties** as AST nodes.

Output Format (example)

```json
{
  "name": "Classes Per File",
  "description": "Analyzes each source file to identify and record all top-
level classes defined",
  "result": {
    "/home/daniel/Workspace/jtmetrics/test/test-src/classes-per-
file/JS/classes.js": {
      "Calculator": [
        {
          "type": "ClassMethod",
          "key": { "type": "Identifier", "name": "foo" },
          "params": [],
          "body": { "type": "BlockStatement", "body": [] }
        },
        {
          "type": "ClassProperty",
          "key": { "type": "Identifier", "name": "bar" },
          "value": { "type": "ArrowFunctionExpression" }
        },
        {
          "type": "ClassProperty",
          "key": { "type": "Identifier", "name": "baz" },
          "value": { "type": "FunctionExpression" }
        }
      ],
      "Logger": [
        {
          "type": "ClassMethod",
          "key": { "type": "Identifier", "name": "foo" }
        }
      ]
    }
  },
  "status": true
}
```

How it works

1. **Dependencies**
    - Requires the `files` metric to know which files are being analyzed.
2. **Class Declarations**
    - Captures **named classes** declared at the top level or exported (`class Foo {}`, `export default class Bar {}`).
    - If no name is provided (e.g., default export), the file name is used as the class name.
    - Ignores invalid cases like inline IIFEs or `class extends class {}`.
3. **Class Expressions**

    - Detects classes assigned to variables (`const Logger = class {}`) or used inside object properties (`{ Printer: class {} }`, `{ "LiteralClassName": class {} }`).

4. **Traversal inside each class**
    - Collects **methods** (`ClassMethod`) and **function-like properties** (`ClassProperty` containing arrow or function expressions).
    - Stores them as an array of AST nodes under each class name.

5. **Post-processing**
    - Cleans up temporary fields (`currentFile`, `dependencies`).
    - Marks the metric as completed by setting `state.status = true`.

Notes

- Classes are grouped **per file**: each file path maps to one or more class names.
- Each class contains an **array of AST nodes** representing its members.
- The metric is designed to **ignore nested or anonymous usage** of classes unless they are explicitly assigned or exported.

Use cases

- Identify how many classes exist per file.
- Analyze **class responsibilities** and complexity by inspecting their collected members.
- Provide a foundation for further metrics (e.g., coupling between classes, method counts, property usage).

== FILE: docs/fileCoupling.doc.md

File Coupling Metric

ID

`file-coupling`

Name

File Coupling

Description

Measures **file-level coupling** by computing each file's **fan-in** (dependent files) and **fan-out** (dependencies).

- **Fan-Out**: files that the current file imports or requires.
- **Fan-In**: files that depend on the current file (other files importing it).

Output Format (example)

```
{
  "name": "File Coupling",
  "description": "Measures file-level coupling by computing each file’s fan-in
(dependent files) and fan-out (dependencies)",
  "result": {
    "/path/to/app.ts": {
      "fanOut": ["/path/to/utils/index.ts"],
      "fanIn": []
    },
    "/path/to/utils/index.ts": {
      "fanOut": [],
      "fanIn": ["/path/to/app.ts"]
    }
  },
  "status": true
}
```

How it works

1. **Dependencies**
    - Requires the `files` metric to know all files in the repository.
2. **Traversal**
    - Detects all **import statements** (`import ... from`), **require calls**, and **TypeScript `import=` declarations**.
    - Resolves relative or absolute paths to real files using `resolveImportPath`.
3. **Fan-Out Recording**
    - Stores each resolved import/require in the `fanOut` array of the current file.
4. **Fan-In Recording**
    - Builds `fanIn` arrays by keeping track of which files import the current file.
5. **Post-processing**
    - Normalizes the results into `{ fanOut: [...], fanIn: [...] }` per file.
    - Cleans temporary state (`currentFile`, `dependencies`) and sets `status` to true.

Notes

- Only considers local files (`.` or `/`)—external packages are ignored.

- Tries `.js`, `.cjs`, `.ts`, `.jsx`, `.tsx`, `.json` extensions and `index.*` in directories.
- Useful to identify **highly dependent files**, **core modules**, or **potential bottlenecks**.

=== FILE: docs/files.doc.md

Files Metric

ID

`files`

Name

Files on Repository

Description

Collects and records all source files in the repository by their path. The final result is a simple array of file paths that were discovered during AST traversal.

Output Format (example)

```json
{
  "name": "Files on Repository",
  "description": "Collects and records all source files in the repository by their path.",
  "result": [
    "/home/daniel/Workspace/jtmetrics/test/test-src/files/JS/fileA.js",
    "/home/daniel/Workspace/jtmetrics/test/test-src/files/JS/subdir/fileB.js",
    "/home/daniel/Workspace/jtmetrics/test/test-src/files/TS/fileA.ts",
    "/home/daniel/Workspace/jtmetrics/test/test-src/files/TS/subdir/fileB.ts"
  ],
  "status": true
}
```

How it works

1. **Traversal step (Program visitor)**
   - On each parsed file, the metric reads `path.node.filePath` and stores it in `state.currentFile`.
   - It adds an entry to `state.result` keyed by that file path (initially assigned an empty object).

2. **Post-processing**

   - Removes the temporary `currentFile` field from state.
   - Transforms `state.result` (an object keyed by file paths) into an **array** of keys: `Object.keys(state.result)`.
   - Filters out keys that are numeric-only (regex `/^\d+$/`) to avoid including numeric-only paths.
   - Sets `state.status = true` to mark completion.

Notes

- The implementation accumulates file paths as object keys during traversal and converts them to an array at the end to produce a clean list.
- The filter `!/^\d+$/.test(k)` is used to exclude numeric-only keys from the final list.
- Paths in `result` are absolute.

Use cases

- Produce a complete inventory of source files in a repository.
- Provide a file list for other metrics that require per-file mapping (e.g., classes-per-file, functions-per-file).

== FILE: docs/functionCoupling.doc.md

Function Coupling Metric

ID

`function-coupling`

Name

Function Coupling

Description

Measures **function-level coupling** by recording **Fan-In** and **Fan-Out** relationships between functions.

- **Fan-Out**: functions called by the current function.
- **Fan-In**: functions that call the current function.

Output Format (example)

```
{
  "name": "Function Coupling",
  "description": "Measures function-level coupling by recording Fan-In and Fan-Out relationships between functions",
  "result": {
    "/path/to/file.js": {
      "foo": {
        "type": "FunctionDeclaration",
        "fan-out": { "bar": 2, "baz": 1 },
        "fan-in": { "qux": 1 }
      },
      "bar": {
        "type": "FunctionExpression",
        "fan-out": { "baz": 1 },
        "fan-in": { "foo": 2 }
      }
    },
    "/path/to/anotherFile.ts": {
      "baz": {
        "type": "ArrowFunctionExpression",
        "fan-out": {},
        "fan-in": { "foo": 1, "bar": 1 }
      }
    }
  },
  "status": true
}
```

How it works

1. **Dependencies**
    - Requires the `functions-per-file` metric to know all functions in the repository.
2. **Traversal**
    - For each function declaration, expression, or arrow function assigned to a variable:
        - Traverse all **CallExpression** nodes inside the function body.
        - Detect if the called function exists in the repository (within files of the same extension).
3. **Fan-Out Recording**
    - For each call found, increment the **fan-out** counter for the caller function.
4. **Fan-In Recording**
    - For each call found, increment the **fan-in** counter for the callee function.
5. **Scope Limitations**
    - Only considers calls to **named functions**.
    - Calls to anonymous inline functions or external libraries are ignored.

6. **Post-processing**
    - Cleans temporary state (`currentFile`, `dependencies`) and marks metric completion.

Notes

- Provides a **per-file and per-function mapping** of coupling relationships.
- Useful to compute metrics like **fan-in/fan-out counts**, detect **highly coupled functions**, or analyze **modularity**.

==
FILE: docs/functionsPerFile.doc.md

Functions Per File Metric

ID

`functions-per-file`

Name

Functions Per File

Description

Records all **named functions** found in each source file.
Each function is stored under its file path and mapped by its name to its **AST node**.

Output Format (example)

```json
{
  "name": "Functions Per File",
  "description": "Records all named functions in each source file, mapping function names to their AST node",
  "result": {
    "/path/to/file.ts": {
      "foo": { "type": "FunctionDeclaration", "id": { "name": "foo" } },
      "bar": { "type": "FunctionExpression" },
      "add": { "type": "ArrowFunctionExpression" }
    },
    "/path/to/file.js": {
      "baz": { "type": "FunctionExpression" },
      "qux": { "type": "FunctionExpression", "id": { "name": "quxNamed" } }
    }
  },
  "status": true
}
```

How it works

1. **Dependencies**
   - Requires the `files` metric to provide the list of files being analyzed.
2. **Function Declarations**
   - Captures standard named functions (`function foo() {}` or `async function bar() {}`).
   - Skips unnamed declarations.
3. **Function Expressions**
   - Detects functions assigned to variables (`const baz = function() {}`).
   - Only included if the variable name exists.
4. **Arrow Functions**
   - Captures named arrow functions when assigned to variables (`const add = (a, b) => a + b`).
   - Inline/anonymous arrows (e.g., `arr.map(x => x * 2)`) are ignored.
5. **Post-processing**
   - Removes temporary state (`currentFile`, `dependencies`).
   - Marks metric completion with `status = true`.

Notes

- Functions are **grouped per file**.
- Each entry in a file's object is keyed by the **function name**.
- The value is the **AST node** of that function (includes parameters, body, async/generator flags, etc.).
- Anonymous inline callbacks are intentionally excluded unless assigned to a named variable.

Use cases

- Detect all **top-level named functions** in a codebase.
- Provide data for metrics like *functions per file count* or *async vs sync usage*.
- Enable downstream analysis of function complexity or parameter patterns.

== FILE: docs/instanceMapper.doc.md

Instance Mapper Metric

ID

`instance-mapper`

Name

Instance Mapper

Description

Walks through **each class method** to identify **instance accesses** (`this.prop`) and **local variable instances**, mapping them to their **constructor types**.

- Detects `new ClassName()` inside:
  - `this.prop = new ClassName()`
  - `const/let varName = new ClassName()`
- Tracks instance properties and local variables for all class declarations and expressions.

Output Format (example)

```
{
  "name": "Instance Mapper",
  "description": "Walks through each class method to identify instance accesses (this.prop and local variables) and map them to their constructor types",
  "result": {
    "/path/to/defaultClass.ts": {
      "defaultClass": {
        "this.foo": "AClass",
        "constFoo": "AClass",
        "letFoo": "AClass",
        "this.bar": "AClass",
        "constBar": "AClass",
        "letBar": "AClass"
      }
    },
    "/path/to/instances.ts": {
      "AClass": {
        "this.fooB": "BClass",
        "constFooB": "BClass",
        "letFooB": "BClass"
      },
      "BClass": {
        "this.fooC": "CClass",
        "constFooC": "CClass",
        "letFooC": "CClass"
      }
    }
  },
  "ignore": true,
  "status": true
}
```

How it works

1. **Dependencies**
    - Operates **per file**, does not depend on other metrics.
2. **Class Traversal**
    - Handles `ClassDeclaration`, `ClassExpression`, and `ExportDefaultDeclaration`.
    - Ignores inline or anonymous classes inside IIFEs or nested class expressions.
3. **Instance Detection**
    - Traverses `ClassMethod` and `ClassProperty` nodes.
    - Detects `NewExpression` assigned to:
        - `this.prop` → mapped as `"this.prop": "ConstructorName"`
        - Local variables (`const`/`let`) → mapped as `"varName": "ConstructorName"`
4. **Mapping**
    - Maps all instances created inside a class method or property to the constructor name.
    - Works for arrow functions and regular function expressions in class properties.
5. **Post-processing**
    - Cleans temporary state (`currentFile`).
    - Sets `status` to `true`.
6. **Notes**
    - Marked with `"ignore": true` (not included in the library output, internal/helper metric for class-coupling metric).
    - Useful for analyzing **instance relationships** and **class-level dependencies** inside code.