Analysis Types 1

In this section:

Pattern-based Analysis

Pattern-based analysis detects constructs in the source code that are known to result in software defects based on programming standards, such as CWE and OWASP. Pattern-based static analysis helps ensure that developers are following coding best practices, unit testing best practices, as well as the organization's development policy.

This and all the following analysis types are performed with a built-in or user-defined test configuration; see Configuring Test Configurations.

Code Duplication Analysis

dotTEST can check for duplicate code to help you improve application design and decrease maintenance costs. During code duplication analysis, the code is parsed into smaller language elements (tokens). The tokens are analyzed according to a set of rules that specify what should be considered duplicate code. There are two types of rules for analyzing tokens:

- Simple rules for finding single token duplicates, e.g., string literals
- Complex rules for finding multiple token duplicates, e.g., duplicate methods or statements

Run the Find Duplicated Code built-in test configuration during analysis to execute code duplicates detection rules:

```
builtin://Find Duplicated Code
```

Metrics Analysis

dotTEST can compute several code metrics, such as code complexity, coupling between objects, or lack of cohesion, which can help you understand potential weak points in the code. Run the Metrics test configuration during analysis to execute metrics analysis rules:

```
builtin://Metrics
```

Metrics analysis results are added to the HTML and XML report files; see Viewing Reports.

Setting Metrics Thresholds

You can set upper and lower boundaries so that a static analysis finding is reported if a metric is calculated outside the specified value range. For example, if you want to restrict the number of logical lines in a project, you could configure the Metrics test configuration so that a finding is reported if the Number of Logical Lines metric exceeds the limit.

The built-in Metrics test configuration includes default threshold values. There are some rules, such as Number of Files (METRIC.NOF), for which thresholds cannot be set.

Metric thresholds can be specified using the following methods:

- By using the test configuration interface in DTP (see "Report Center> Test Configurations> Editing Test Configurations> Metrics Tab" in the Development Testing Platform user manual for details).
- By editing the Metrics test configuration using the interface in an IDE (see Creating Custom Test Configurations).
- By manually editing the test configuration file:

  1. Duplicate the built-in Metrics test configuration (`[INSTALL_DIR]/configs/builtin`) to the user configurations directory (`[INSTALL_DIR]/configs/user`).

  2. Open the duplicate configuration in an editor and set the `[METRIC.ID].ThresholdEnabled` property to `true`.

3. Configure the lower and upper boundaries in the `[METRIC.ID].Threshold` property according to the following format: `[METRIC.ID].Threshold=l [lower boundary value] g [upper boundary value]`

4. Save the test configuration and run the analysis using the custom metrics test configuration.

Flow Analysis

Flow Analysis is a type of static analysis technology that uses several analysis techniques, including simulation of application execution paths, to identify paths that could trigger runtime defects. Defects detected include use of uninitialized memory, null pointer dereferencing, division by zero, memory and resource leaks.

Since this analysis involves identifying and tracing complex paths, it exposes bugs that typically evade static code analysis and unit testing, and would be difficult to find through manual testing or inspection.

Flow Analysis' ability to expose bugs without executing code is especially valuable for users with legacy code bases and embedded code (where runtime detection of such errors is not effective or possible).

Run one of the Flow Analysis test configurations during analysis to execute flow analysis rules:

```
builtin://Flow Analysis Fast
builtin://Flow Analysis Standard
builtin://Flow Analysis Aggressive
```

Configuring Depth of Flow Analysis

Flow Analysis builds paths through the analyzed code to detect different kinds of problems. Since the analysis of all possible paths that span through the whole application may be infeasible, you can set up the desired level of depth of analysis. A deeper analysis will result in more findings, but the performance will be slower and the memory consumption will increase slightly.

You can specify the depth of analysis by using the test configuration interface in DTP. Go to **Report Center> Test Configurations> Static Analysis> Flow Analysis Advanced Settings> Performance> Depth of analysis** and choose one of the following options by selecting a radio button:

- **Shallowest (fastest)**: Finds only the most obvious problems in the source code. It is limited to cases where the cause of the problem is located close to the code where the problem occurs. The execution paths of violations found by this type of analysis normally span several lines of code in a single function. Only rarely will they span more than 3 function calls.
- **Shallow (fast)**: Like the "Shallowest" analysis type, finds only the most obvious problems in the source code. However, it produces a greater overall number of findings and allows for examination of somewhat longer execution paths.
- **Standard**: Finds many complicated problems with execution paths containing tens of elements. The standard analysis goes beyond shallow analysis and also looks for more complicated problems, which can occur because of bad flow in a single function or due to improper interaction between different functions in different parts of the analyzed project. Violations found by this type of analysis often reveal non-trivial bugs in the analyzed source code and often span tens of lines of code.
- **Deep (slow)**: Allows for detection of a greater number of problems of the same complexity and nature as those defined for "Standard" depth. This type of analysis is slower than the standard one.
- **Thorough (slowest)**: Finds more complicated problems. This type of analysis will perform a thorough scan of the code base; this requires more time, but will uncover many very complicated problems whose violation paths can span more than a hundred lines of code in different parts of the scanned application. This option is recommended for nightly runs.

The depth of Flow Analysis is set to **Standard** by default.

Setting Timeout Strategy

Apart from the depth of analysis, Flow Analysis uses an additional timeout guard to ensure the analysis completes within a reasonable time. An appropriate strategy can be set by using the test configuration interface in DTP. Go to **Report Center> Test Configurations> Static Analysis> Flow Analysis Advanced Settings> Performance> Strategy for Timeouts** and choose one of the follow-ing options by selecting a radio button:

- **time**: Analysis of the given hotspot is stopped after spending the defined amount of time on it. Note: in some cases, using this option can result in a slightly unstable number of violations being reported.
- **instructions**: Analysis of the given hotspot is stopped after executing the defined number of Flow Analysis instructions.Note: to determine the proper number of instructions to be set up for your environment, review information about timeouts in the Setup Problems section of the generated report.
- **off**: No timeout. Note: using this option may require significantly more time to finish the analysis.

The default timeout option is **time** set to 60 seconds. To get information about the Flow Analysis timeouts that occurred during the analysis, review the Setup Problems section of the report generated after the analysis.

Running Flow Analysis with Swapping of Analysis Data Enabled

In this mode, analysis data is written to disk. Swapping of analysis data uses the same persistent storage and is done in a similar process as incremental analysis. If the analysis is run on a large project, the analysis data that represents a semantical model of the analyzed source code may consume all the memory available for running Flow Analysis. If this occurs, Flow Analysis will remove from memory parts of the analysis data that are not currently necessary and reread it from disk later.

In general, we recommend running in a large JVM heap configured with the Xmx JVM option. This is to minimize swapping, which results in greater performance. If sufficient memory is available, swapping of analysis data may be disabled, which may speed up code analysis.

You can enable or disable the mode by using the test configuration interface in DTP:

Enable swapping of analysis data to disk:

Disabled by default. If this option is disabled, it may result in faster analysis, if you are running Flow Analysis analysis on small to moderate size projects that do not require a lot of memory or when plenty of memory is available (for example, for 64-bit systems).

Configuring Verbosity of Flow Analysis

You can configure the following options by using the test configuration interface in DTP:

- **Do not report violations when cause cannot be shown**: Determines whether Flow Analysis reports violations where causes cannot be shown. Some Flow Analysis rules require that Flow Analysis checks all the possible paths leading to a certain point and verifies that a certain condition is met for all those paths. In such cases, a violation is associated with a set of paths (whereas in simple cases, a violation is represented by only one path). All of the paths in such a violation end with the violation point common to all the paths in the violation. However, different paths may start at different points in code. The beginning of each path is a violation cause (a point in code which stipulates a violation of a certain condition later in the code at the violation point). If a multipath violation's different paths have different causes, Flow Analysis will show only the violation point (and not the violation causes).
  Violations containing only the violation point may be difficult to understand (compared to regular cases where Flow Analysis shows complete paths starting from violation causes and leading to violation points). That's why we provide an option to hide violations where the cause cannot be shown.
- **Do not show more than one violation per point:** Restricts reporting to one violation (for a single rule) per violation point. For example, one violation will be reported when Flow Analysis detects a potential null dereference with multiple sources of the null value. When verbosity is set to this level, Flow Analysis performance is somewhat faster.

Null-checking Methods

The **Null-checking methods** option allows you to specify the expected return value when a null parameter is passed to a method. This reduces false positives and excessive paths that would normally be built when the return value for null variables are unknown.

Select the **Enabled** checkbox and provide the following information:

- **Fully-qualified type name (wildcard)**: the fully qualified name of the type that contains the method.
- **Method name (wildcard)**: the name of the method.
- **Returned value when null**: the value that should be returned when a null parameter is passed to the method.
- **+ definitions in subclasses**: indicates whether the null-checking functions definitions in sub-classes should be considered null-checking functions as well.

Null-checking Methods Example

Flow Analysis will analyze methods in the analysis scope. Null-checking method parameterizations are required when the methods are out of the scope of the current analysis (e.g. third-party libraries). The classes `Account` and `AccountManager` in the following example are defined in a different assembly and are outside the scope of the analysis. The class `AccountManager` uses static methods to check `Account` in following way:

```
public class Account
    {
        public int Balance { get; set; }
    }

    public class AccountManager
    {
        public static bool IsNullOrEmpty(Account account)
        {
            return account == null || account.Balance == 0;
        }
    }
```

And there is the following class, which is analyzed by Flow Analysis's BD.EXCEPT.NR rule:

```
public class SomeClass
{
    void SomeMethod()
    {
        Account account = null;
        //some other actions
        if (!AccountManager.IsNullOrEmpty(account))
        {
            Console.WriteLine(account.Balance);
        }
    }
}
```

Flow Analysis assumes that the return value for the method `IsNullOrEmpty()` is unknown. This is because the method is not in the analysis scope. Flow Analysis will analyze the if branch and find a violation. Calling the `IsNullOrEmpty()` method, however, returns true when the variable is null.

By adding `AccountManager.IsNullOrEmpty()` to the list of methods in the Null-checking Methods parameterization with the specified return value, Flow Analysis will not report a violation if the tracked variable is passed to this method. This is because the `if` branch will not be analyzed, resulting in an avoided false positive.

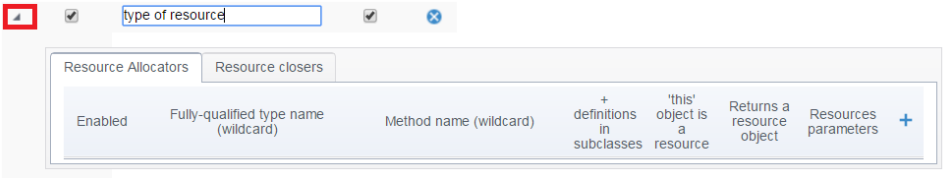
Null-checking Methods Restrictions

Methods added to this parameterization should be static methods that have a primitive bool return value. Additionally, null-checking methods can have only one input parameter. These restrictions are managed to avoid excessive complications in parameterization and ensuring that there are no other variables that could affect the result of the null-checking method.

Specifying Resources

The **Resources** tab allows you to define which resources the BD.RES category (Resources) rules should check. These rules check for the correct usage of all resources that are defined and enabled on this tab.

1. Specify the Type of resource.
2. Select the Enabled checkbox.
3. If appropriate/desired, disable the Do not report leaks at termination option.
4. Click the arrow to expand the Resource Allocators and Resource Closers tabs and complete the tables that open with the information about allocators and closers. Details about completing these tabs are provided below.



Configuring Resource Allocators

The **Resource** allocators table can be completed with the descriptors of methods that can produce a resource. The table has the following columns:

- **Enabled**: specifies whether the allocator should be considered during analysis.
- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the method is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Method name (wildcard)**: the name of the allocating method. '*' can be used to denote any number of any symbols.
- **Resource parameters**: specifies that the method allocates a resource in one or more of its parameters. In this case, either specify a 1-based number of the parameter that is allocated by the method, or use '*' to denote that all of the parameters are allocated.
- **+ definitions in subclasses**: a check box that indicates whether the definitions (of methods with the given name) in subclasses should be considered allocators as well. Note that this applies to both instance and static methods.
- **"this" object is a resource**: a check box that indicates that the method allocates a resource in the object on which the method is called.
- **Returns a resource object**: a check box that indicates that the method returns an allocated resource.

It is common that allocation methods return an error code to indicate allocation failure. When an allocation method returns a resource, a NULL value normally indicates an allocation failure. When Flow Analysis is looking for resource leaks, it needs to understand if allocation succeeded or failed; this helps it report only missing calls to deallocation methods on paths where allocation actually occurred. In cases where a resource allocator method returns a resource, Flow Analysis assumes that the resource is successfully allocated if the returned value is not NULL.

Configuring Resource Closers

The **Resource** closers table can be completed with the descriptors of methods that can close a resource. The table has the following columns:

- **Enabled**: specifies whether the closer should be considered during analysis.

- **Fully-qualified type name or namespace (wildcard)**: the fully-qualified name of the type or namespace where the method is declared. Use '*' if you want to describe a function declared in any type or namespace, or a global function declared outside of any type.
- **Method name (wildcard)**: the name of the closing method. '*' can be used to denote any num-ber of any symbols.
- **+ definitions in subclasses**: a check box that indicates whether the definitions (of methods with the given name) in subclasses should be considered closers as well. Note that this applies to both instance and static methods.
- **"this" object is a resource**: a check box that indicates that a resource in the object on which the method is called is closed.
- **Resource parameters**: specifies that a resource in one or more of its parameters is closed. In this case, either specify a 1-based number of the parameter that is closed by the method, or use '*' to denote that all of the parameters are allocated.