# MSC56-J. Detect and remove superfluous code and values

Superfluous code and values may occur in the form of dead code, code that has no effect, and unused values in program logic.

Code that is never executed is known as *dead code*. Typically, the presence of dead code indicates that a logic error has occurred as a result of changes to a program or to the program's environment. Dead code is often optimized out of a program during compilation. However, to improve readability and ensure that logic errors are resolved, dead code should be identified, understood, and removed.

Code that is executed but fails to perform any action, or that has an unintended effect, most likely results from a coding error and can cause unexpected behavior. Statements or expressions that have no effect should be identified and removed from code. Most modern compilers can warn about code that has no effect.

The presence of unused values in code may indicate significant logic errors. To prevent such errors, unused values should be identified and removed from code.

## Noncompliant Code Example (Dead Code)

This noncompliant code example demonstrates how dead code can be introduced into a program [Fortify 2013]:

```
public int func(boolean condition) {
  int x = 0;
  if (condition) {
    x = foo();
    /* Process x */
    return x;
  }
  /* ... */
  if (x != 0) {
    /* This code is never executed */
  }
  return x;
}
```

The condition in the second if statement, `(x != 0)`, will never evaluate to `true` because the only path where `x` can be assigned a nonzero value ends with a `return` statement.

## Compliant Solution

Remediation of dead code requires the programmer to determine not only why the code is never executed but also whether the code should have been executed, and then to resolve that situation appropriately. This compliant solution assumes that the dead code should have executed, and consequently, the body of the first conditional statement no long ends with a return.

```
int func(boolean condition) {
  int x = 0;
  if (condition) {
    x = foo();
    /* Process x */
  }
  /* ... */
  if (x != 0) {
    /* This code is now executed */
  }
  return 0;
}
```

## Noncompliant Code Example (Dead Code)

In this example, the `length()` function is used to limit the number of times the function `string_loop()` iterates. The condition of the `if` statement inside the loop evaluates to `true` when the current index is the length of `str`. However, because `i` is always strictly less than `str.length()`, that can never happen.

```
public int string_loop(String str) {
  for (int i=0; i < str.length(); i++) {
    /* ... */
    if (i == str.length()) {
      /* This code is never executed */
    }
  }
  return 0;
}
```

## Compliant Solution

Proper remediation of the dead code depends on the intent of the programmer. Assuming the intent is to do something special with the last character in `str`, the conditional statement is adjusted to check whether `i` refers to the index of the last character in `str`.

```
public int string_loop(String str) {
  for (int i=0; i < str.length(); i++) {
    /* ... */
    if (i == str.length()-1) {
      /* This code is now executed */
    }
  }
  return 0;
}
```

## Noncompliant Code Example (Code with No Effect)

In this noncompliant code example, the comparison of `s` to `t` has no effect:

```
String s;
String t;

// ...

s.equals(t);
```

This error is probably the result of the programmer intending to do something with the comparison but failing to complete the code.

## Compliant Solution

In this compliant solution, the result of the comparison is printed out:

```
String s;
String t;

// ...

if (s.equals(t)) {
  System.out.println("Strings equal");
} else {
  System.out.println("Strings unequal");
}
```

## Noncompliant Code Example (Unused Values)

In this example, `p2` is assigned the value returned by `bar()`, but that value is never used:

```
int p1 = foo();
int p2 = bar();

if (baz()) {
  return p1;
} else {
  p2 = p1;
}
return p2;
```

## Compliant Solution

This example can be corrected in many different ways depending on the intent of the programmer. In this compliant solution, `p2` is found to be extraneous. The calls to `bar()` and `baz()` could also be removed if they do not produce any side effects.

```
int p1 = foo();

bar(); /* Removable if bar() lacks side effects */
baz(); /* Removable if baz() lacks side effects */

return p1;
```

## Applicability

The presence of dead code may indicate logic errors that can lead to unintended program behavior. The ways in which dead code can be introduced into a program and the effort required to remove it can be complex. As a result, resolving dead code can be an in-depth process requiring significant analysis.

In exceptional situations, dead code may make software resilient to future changes. An example is the presence of a default case in a `switch` statement even though all possible switch labels are specified (see MSC57-J. Strive for logical completeness for an illustration of this example).

It is also permissible to temporarily retain dead code that may be needed later. Such cases should be clearly indicated with an appropriate comment.

The presence of code that has no effect can indicate logic errors that may result in unexpected behavior and vulnerabilities. Unused values in code may indicate significant logic errors.

Unused classes, methods, and variables that are part of an exported library do not violate this guideline.

Code and values that have no effect can be detected by suitable static analysis.

## Automated Detection

| Tool | Version | Checker | Description |
|---|---|---|---|
| SonarQube | 6.7 | **S1854** | |

## Bibliography


| | |
|---|---|
| [Coverity 2007] | *Coverity Prevent User's Manual (3.3.0)* |
| [Fortify 2013] | Code Quality, "Dead Code" |