GoQu

GOSU QUALITY ASSURANCE

sollers
CONSULTING

Keep your code clean

As our business digitise, code quality becomes ever more significant.

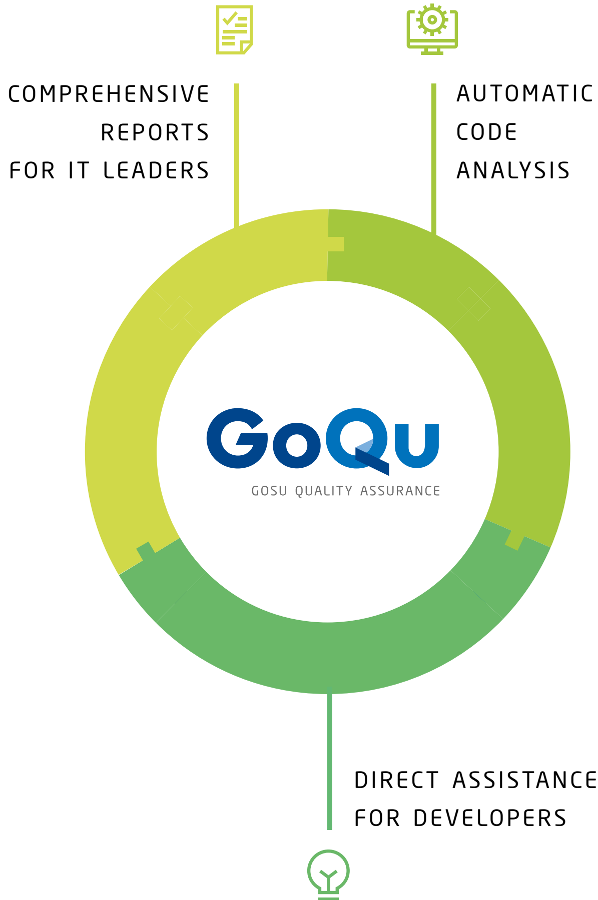
Did you know that due to software bug Knight Capital Group **lost $440 million** in 45 minutes?

As a consulting company Sollers is expected to provide the **highest quality software** to our clients.

Insurance core systems can harbour as many as one hidden bug, smell or vulnerability per every three lines of code.

To prevent it and ensure reliability of our implementations, Sollers developed GoQu-

a tool for quality assurance dedicated to Guidewire products.



GoQu suite

assists developers with code analysis directly in Guidewire Studio, preventing bad code from ever being written.

enables code quality monitoring of your Guidewire Centers, providing IT leadership with transparent reports.

combines the features of GoQu Hub and Web to be used in portals world.

We selected 9 common issues

which can impact code quality:

HIDDEN TIME BOMBS

- Magic numbers should not be used (p. 10)
- „Switch" statements should end with „default" clause (p. 12)
- Two branches in a conditional structure should not have the same entering condition (p. 14)

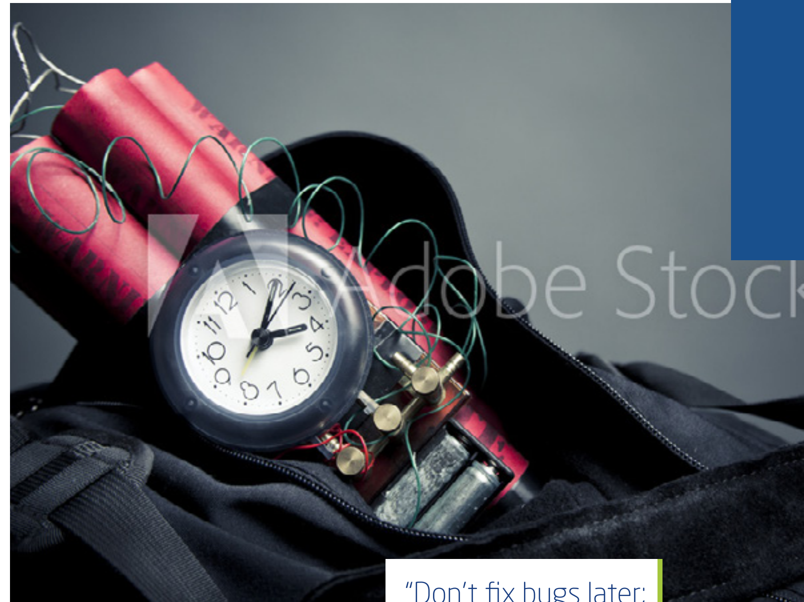
ERRORS

- Static fields should not be updated in constructors (p. 18)
- Excessive imports should be removed (p. 20)
- Throwable and Error should not be caught (p. 22)

COMPLEXITY

- Avoid n nested loop constructs (p. 26)
- Expressions should not be too complex (p. 28)
- Complexity of methods should not be too high (p .30)

Hidden time bombs



"Don't fix bugs later; fix them now."

STEVE MAGUIRE

Magic numbers should not be used

Hardcoded values can lead to significant issues.

Example: current tax rate could be hardcoded in software, potentially **losing company significant amount of money and risking non-compliance when regulations change.**

Use private static final field of a class.

Compliant Code:

```
public class TaxCalculator {

    private static final var TAX_MULTIPLIER = 0.25

    public function calculateTax (amount : double ):
    double {
    return amount * TAX_MULTIPLIER
   }
}
```

Noncompliant Code:

```
public class TaxCalculator {

    public function calculateTax(amount : double):
    double {
    return amount * 0.25
    }
  }
```

„Switch" statements should end with „default" clause

Code without default execution path can lead to situation when no defined action is assigned to a specific scenario.

This can result in critical error of the software - potentially reasulting in business interruption.

The clause should either take appropriate action, or contain a suitable comment as to why no action is taken.

Default clause should always exist, even if it is the enum or typelist used as key and all variants are covered, as it could be always extended by someone so your code won't cover this case and can cause unexpected behavior.

Compliant Code:

```
switch (parameter) {
  case 0:
  doSomething()
  break
  case 1:
  doSomethingElse()
  break
  default:
  handleError()
  break
}
```

Noncompliant Code:

```
switch (parameter) { // missing default clause
  case 0:
  doSomething()
  break
  case 1:
  doSomethingElse()
  break
}
```

Two branches in a conditional structure should not have the same entering condition

Duplicating an entering condition automatically leads to dead code (section in the code that is never used). Usually, this is due to a copy/paste error.

At best, it's simply just a dead code and at **worst, it's a bug that is likely to induce further bugs as the code is maintained, and could lead to unexpected behavior of the software.**

A chain of if/else if statements is evaluated from top to bottom. At most, only one branch will be executed: the first one with a condition that evaluates to true.

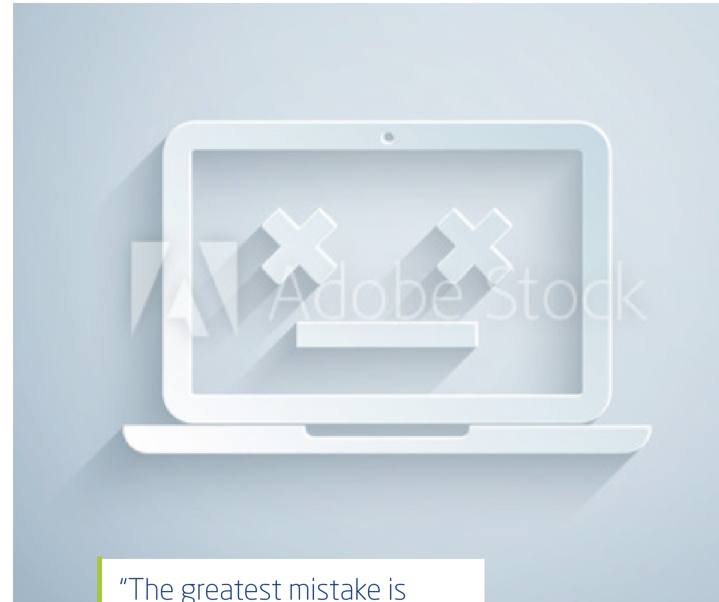
Compliant Code:

```
if (parameter == 1) {
  openWindow()
} else if (parameter == 2) {
  closeWindow()
} else if (parameter == 3) {
  moveWindowToTheBackground()
}
```

Noncompliant Code:

```
if (parameter == 1) {
  openWindow()
} else if (parameter == 2) {
  closeWindow()
} else if (parameter == 1) { // Noncompliant
  moveWindowToTheBackground()
}
```

Errors



"The greatest mistake is
to imagine that we never err."

THOMAS CARLYLE

Static fields should not be updated in constructors

It may override the expected values with wrong ones. **As a consequence it may produce unexpected behavior and may interrupt internal processes.**

These fields can be referenced before creating an instance of the class. Setting the value in the constructor will be done later or not at all.

Compliant Code:

```
public class Person {
   private var _dateOfBirth : Date
   private static var _expectedFingers : int = 10
   construct(birthday : Date) {
       _dateOfBirth = birthday
   }
}
```

Noncompliant Code:

```
public class Person {
   private static var _dateOfBirth : Date
   private static var _expectedFingers : int
   construct(birthday : Date) {
        _dateOfBirth = birthday  // Noncompliant -
                                //now everyone has
                               //this birthday
       _expectedFingers = 10  // Noncompliant
  }
}
```

Excessive imports should be removed

It may result in the usage of wrong or outdated version of the library - **adversely impacting the release and deployment.**

Compliant Code:

```
package my.company.mypackage
uses com.sample.animals.Cat
uses com.sample.plants.flower
class Garden {
     function makeNoise(cat : Cat) {
      cat.makeSound()
     }
     function lookupLatinName(flower : Flower): String {
     // ...
     }
 // ...
}
```


Noncompliant Code:

```
package my.company.mypackage
uses java.lang.String;
uses com.sample.animals.Cat
uses com.sample.animals.Dog
uses com.sample.plants.*
class Garden {
    function makeNoise(cat : Cat) {
     cat.makeSound()
     }
    function lookupLatinName(flower : Flower) : String {
     // ...
     }

 // ...
}
```

Throwable and Error should not be caught

Catching NullPointerException should also be avoided, as stack trace will be lost.

It hides the origin of the original error and reports something else in exchange. To get to the bottom of the pro-blem, in some cases company would have to **manually check the entire codebase.**

Throwable is the superclass of all errors and exceptions in Java. Error is the superclass of all errors, which are not meant to be caught by applications. Catching either Throw-able or Error will also catch OutOfMemoryError and Inter-nalError , from which an application should not attempt to recover.

Exception re-thrown catch clauses should also provide some additional handling apart from throw statement.

Catching NullPointerException should also be avoided, as stack trace will be lost. Most situation in which NullPointerException is explicitly caught can be converted to a null test, and any behavior being carried out in the catch block can easily be moved to the "is null" branch of the con-ditional.

Compliant Code:

```
try {
      //...
} catch (exception : MyCustomException) {
      doSomeErrorHandling(exception)
}
try {
      //...
} catch (exception : Exception) {
      doSomeInitialErrorHandling(exception)
throw exception
}
```

Noncompliant Code:

```
try {
      //...
} catch (throwable : Throwable) {
      //...
}
try {
      //...
} catch (error : Error) {
      //...
}
```

Complexity

"Any intelligent fool can make things bigger and more complex... It takes a touch of genius – and a lot of courage to move in the opposite direction."

E. F. SCHUMACHER

Avoid n nested loop constructs

Unnecessarily nested loops often form **inefficient code** or code that **does not produce correct results.** They can also cause **performance issues**.

Issue indicates that you should rethink if your solution is designed well. Sometimes nested loops are really needed, for example to get some information. However, sometimes they can be dangerous and cause performance issues. Unnecessarily nested loops often form inefficient code or code that does not produce correct results.

The complexity limit can be customized (default limit is set to 4).

Compliant Code:

```
var numbers = new int[]{1, 2, 3, 4, 3}
for (a in numbers) { // Compliant - depth = 1
   for (b in numbers) { // Compliant - depth = 2
     for (y in numbers index m) { // Compliant - depth = 3
       for (z in numbers index n) { // Compliant - depth = 4
         if (m != n and y == z) {
            doSomething(z)
         }
       }
     }
   }
 }
```

Noncompliant Code:

```
var numbers = new int[]{1, 2, 3, 4, 3}
for (a in numbers) { // Compliant - depth = 1
 for (b in numbers) { // Compliant - depth = 2
  for (c in numbers) { // Compliant - depth = 3
   for (y in numbers index m) { // Compliant - depth = 4
    for (z in numbers index n) { // Noncompliant -
                                 //depth = 5, exceeding
                                 //the limit
      if (m != n and y == z) {
         doSomething(z)
      }
     }
    }
   }
  }
}
```

Expressions should not be too complex

Using high number of logical and other operators can result in **limited readability** of the code and **complicates maintenance** of the software.

This makes future upgrades far more difficult.

The complexity of an expression is defined by the number of logical operators (and, or, &&, ||, &, |) and ternary operators (condition ? ifTrue : ifFalse) it contains. A single expression's complexity should not become too high to keep the code readable. Logical operators limit can be customized (default is set to 5).

Compliant Code:

```
if ((checkFirstCondition() or checkSecondCondition())
and checkThirdCondition()) {
       //...
}
```

Noncompliant Code:

```
if (((condition1 and condition2) or (condition3 and
condition4)) and condition5) {
       //...
}
```

Complexity of methods should not be too high

High number of linearly independent paths in a software can **affect negatively performance, maintainability of the code and easiness to debug.**

Cyclomatic complexity is a quantitative measure of the number of linearly independent paths through a program's source code.

Compliant Code:

```
public function isVehicleEligibleForSpecialInsurance(vehicle:VehicleDTO):
boolean{
   var eligible: boolean
   if (isVehicleOfEligibleType(vehicle)) { // if: +1 complexity
        eligible = true
   } else {
        switch (vehicle.Fuel) {
              case DIESEL: // case: +1 complexity
              eligible = verifyEligibilityForDiesel(vehicle.EngineData)
              break
              case LPG: // case: +1 complexity
              eligible = verifyEligibilityForLPG(vehicle.EngineData)
              break
              case GASOLINE: // case: +1 complexity
              eligible = haveEnoughModifications(vehicle)
              break
              default: // default: +1 complexity
              eligible = true
              break
       }
   }
   return eligible
} // complexity in total: 5
```

We recommend complexity to reach level 5 at most, but saw code including complexities reaching as high as 250.

GoQu
GOSU QUALITY ASSURANCE

BY sollers CONSULTING

This was just a small sample of infractions, which can be identified by GoQu. Our tool includes already **more than 120** dedicated rules for bugs, smells and vulnerabilities, with new functionalities being added monthly.

If you would you like to know more details about GoQu and trial it for free - contact us at:

GoQu@sollers.eu

sollers.eu