CHAPTER V

VALIDATING DESIGN PRINCIPLE VIOLATION CHECK DEFINITIONS

Chapter V validates design principle violation check definitions. All violation check definitions are confirmed their correctness by using them to detect design principle violations in code samples written from the original examples and the quotations of the principle sources.

Code samples are taken from four sources: object-oriented design heuristics, bad smells for object-oriented refactorings, bad smells for aspect-oriented refactorings, and examples in Section 3.2 [36, 15, 40]. Violation checks for these code samples are described below. The results show that all flaws in systems are exposed by the design principle violation check definitions. Thus, all definitions are valid and are passed. Some definitions are valid under conditions such as the violation check definitions for *Long Method* can detect violations in methods containing more than thirteen lines of code.

Guideline 1: Don't make the core concern unused by replacing the entire of its context with the crosscutting concern.

Source code:

```
public aspect A1 {
        pointcut callDo2(): call (void source39.C.do2());
        void around(): callDo2(){
                do2a();
        }
        public void do2a(){
                System.out.println("Do2a");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *AAVNCP* be the set of around advices of U which did not call *proceed()*.

The principle is violated if

$$AAVNCP \neq \emptyset$$

Measurement example:

Aspect A1

AAVNCP = {around():callDo2()} // Detected: aspect A1 violates Guideline 1.

Guideline 2: Do suitably use wildcard operators for each captured join point to increase the pointcut readability and to decrease the unintended join point capture.

Source code:

```
public aspect A {
        pointcut p(): execution(* Account*.*());
        before(): p(){
                System.out.println("Check Authorization");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *P* be the set of pointcuts of U.

Let *CJ* be the set of captured join points of a pointcut *p* where $p \in P$.

Let $ADWCJ_{pcj}$ be the set of asterisk and dotted line wildcard operators of a captured join point *cj* of *p* where $cj \in CJ$.

The principle is violated if

$$\exists p \ \exists cj \ [\ |ADWCJ_{pcj}|>2]$$

Measurement example:

Aspect A

P = {p()}

CJ = {execution(* Account*.*())}

$ADWCJ_{p()execution(* Account*.*())}$ = {*1, *2, *3}

$|ADWCJ_{p()execution(* Account*.*())}|$ = 3 // Detected: aspect A violates Guideline 2.

Guideline 3: Do define a concise pointcut for capturing sibling join points to decrease the accidental join point miss when a new class is added to the hierarchy.

Source code:

```
public class Account {


}
public class FixedAccount extends Account{
        public FixedAccount(){
                System.out.println("FixedAccount constructor");
        }
}
public class SavingAccount extends Account{
        public SavingAccount(){
                System.out.println("SavingAccount constructor");
        }
}
public class CurrentAccount extends Account{
        public CurrentAccount(){
                System.out.println("CurrentAccount constructor");
        }
}
public aspect A {
        pointcut p(): execution(CurrentAccount.new())|| execution(SavingAccount.new())||
        execution(FixedAccount.new());
        before(): p(){
                System.out.println("Check Authorization");
        }
}
```

Violation check expressions:

Consider a unit U.

Let P be the set of pointcuts of U.

Let ASU_p be the set of all sibling units which their similar methods are captured by a pointcut p where $p \in P$.

The principle is violated if

$$\exists p \, [ASU_p \neq \emptyset]$$

Measurement example:

Aspect A

P = {p()}

ASU_{p0} = {CurrentAccount, SavingAccount, FixedAccount} // Detected: aspect A violates Guideline 3.

Guideline 4: Don't use an aspect privilege to leak the secret of other types.

Source code:

```
public class StepUp {
        private int i = 0;
        void setI(int a) {
                if (a > 20)    // condition: i <= 20
                        this.i = 20;
                else
                        this.i = a; }

        int incI() { return this.i+1; }
}
privileged aspect JumpStart {
        before(StepUp s): call(int StepUp.incI()) && target(s) {
                s.i = 21; }
}
public class Test {
        public static void main(String args[]){
                StepUp s = new StepUp();
                s.setI(20);
                System.out.println(s.incI()); // should print 21
                s.setI(21);
                System.out.println(s.incI()); // still print 21
        }
}
```

Violation check expressions:

Consider a privilege unit PU.

Let FT be the set of functions (methods/ intertype-methods/ advices) of PU.

Let $PVMOU_{ft}$ be the set of private members (fields/ inter-type fields/ methods/ inter-type methods) of other units except PU which are used by *ft* where $tf \in FT$. The principle is violated if

$$\exists ft[\, PVMOU_{ft} \neq \emptyset]$$

Measurement example:

Aspect JumpStart

PU = JumpStart

FT = {before(StepUp s): call(int StepUp.incI()) && target(s)}

$PVMOU_{before(StepUp\ s):\ call(int\ StepUp.incI())\ \&\&\ target(s)}$ = **{Stepup.i}** // Detected: aspect JumpStart violates Guideline 4.

Guideline 5: Do create an advantageous aspect which crosscuts at least two points in the core concern.

Source code:

```
public class MessageCommunicator {
   public static void deliver(String message) {
         System.out.println(message);
   }

   public static void deliver(String person, String message) {
         System.out.print(person + ", " + message);
   }
}
public class Test {
   public static void main(String[] args) {
         MessageCommunicator.deliver("Wanna learn AspectJ?");

   }
}
public aspect MannersAspect {
   pointcut deliverMessage()
         : call(* MessageCommunicator.deliver(..));

   before() : deliverMessage() {
```

```
        System.out.print("Hello! ");
    }
}
```

Violation check expressions:

Consider a unit U.

Let *P* be the set of pointcuts of U.

Let J_p be the set of corresponding join points of a pointcut *p* where $p \in P$.

Let *AV* be the set of advices of U. Let *WESD* be the set of warnings, errors, and soft declarations of U.

Let *AP* be the set of abstract pointcuts of U.

Let *PD* be the set of parent declarations of U.

Let *PDD* be the set of precedence declarations of U.

Let *F* be the set of fields of U.

Let *IF* be the set of inter-type fields of U.

Let *IM* be the set of inter-type methods of U.

Let *M* be the set of methods of U.

Let *max(N)* be the function to determine the max number of *N* where *N* is the set of counting numbers.

The principle is violated if

$$(F \cup IF \cup M \cup IM \cup P \cup AV \neq \emptyset) \wedge max(|J_p|, |AV|, |WESD|) + |AP| + |PD| + |PDD| + |IF| + |IM| < 2$$

Measurement example:

Aspect MannersAspect

$F = \{\}$

$IF = \{\}$

$M = \{\}$

$IM = \{\}$

$P = \{$ deliverMessage()$\}$

$AV = \{$ before() : deliverMessage()$\}$

$J_{deliverMessage()}$ = { MessageCommunicator.deliver("Wanna learn AspectJ?");}

AP = {}

PD = {}

PDD = {}

($F \cup IF \cup M \cup IM \cup P \cup AV$ = {deliverMessage(),before() : deliverMessage()}) $\wedge$ max (1, 1, 0) + 0 + 0 + 0 + 0 + 0 = 1 // Detected: aspect MannerAspect violates Guideline 5.

Guideline 6: Do localize all crosscutting contexts within a set of corresponding aspects.

Source code:

```
public class C2 {
        public void do4(){
                System.out.println("Do4");
        }
        public static void printBlank(){
                System.out.println("");
        }
        public void do5(){
                System.out.println("Do5");
        }
}
public aspect A1 {
        pointcut callDo2(): call (void source46.C1.do2());
        after(): callDo2(){
                System.out.println("");
        }

}
```

Violation check expressions:

Consider a unit U.

Let *M* be the set of methods of U.

Let *IM* be the set of inter-type methods of U.

Let *OFT* be the set of other functions (methods, inter-type methods, and advices) of all units except the given method or the given inter-type method *mim* where $mim \in (M \cup IM)$.

Let ASF_{mim} be the set of adjacent statements of *mim*.

Let ASF_{oft} be the set of adjacent statements of *oft* where $oft \in OFT$.

The principle is violated if

$$\exists\, mim\, \exists\, oft\, (ASF_{mim} \cap ASF_{oft} \neq \emptyset)$$

Measurement example:

Class C2

M = {do4(), printBlank(), do5}

IM = {}

$ASF_{do4()}$ = { ØSystem.out.println("Do4"); , System.out.println("Do4");Ø}

$ASF_{printBlank()}$ = { Ø System.out.println(""); , System.out.println("");Ø}

$ASF_{do5()}$ = { ØSystem.out.println("Do5"); , System.out.println("Do5");Ø}

$ASF_{after(): callDo2()}$ = { Ø System.out.println(""); , System.out.println("");Ø}

$ASF_{do4()} \cap ASF_{printBlank()} = \emptyset$

$ASF_{do4()} \cap ASF_{do5()} = \emptyset$

$ASF_{do4()} \cap ASF_{after(): callDo2()} = \emptyset$

$ASF_{printBlank()} \cap ASF_{do5()} = \emptyset$

$ASF_{printBlank()} \cap ASF_{after(): callDo2()}$ = { ØSystem.out.println("Do4"); , System.out.println("Do4");Ø} // Detected: class C2 violates Guideline 6.

$ASF_{do5()} \cap ASF_{after(): callDo2()} = \emptyset$

Guideline 7: Do try to separate one set of aspects to be independent of another.

Source code:

```
public aspect A1 {
        pointcut callDo2(): call (void source47.C.do2());
        before(): callDo2(){
                A2.do2a();
                do2b();
```

```
        }
        public void do2b(){
                System.out.println("Do2b");
        }
}
public aspect A2 {
        pointcut callDo2(): call (void source47.C.do2());
        after(): callDo2(){
                do2c();
        }
        public static void do2a(){
                System.out.println("Do2a");
        }
        public static void do2c(){
                System.out.println("Do2c");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *M* be the set of methods of U.

Let *IM* be the set of inter-type methods of U.

Let *AV* be the set of advices of U.

Let *MIMO* be the set of methods and inter-type methods of other aspect units except U.

Let $FCMIM_{mimo}$ be the set of functions (methods/ inter-type methods/ advices) of other units calling to a method or an inter-type methods *mimo* where *mimo* ∈ *MIMO*.

The principle is violated if

$$\exists\, mimo[\ (M \cup IM \cup AV) \cap FCMIM_{mimo} \neq \emptyset]$$

Measurement example:

Aspect A1

M = { do2b()}

IM = {}

AV = { before(): callDo2()}

MIMO = { do2a(),do2c()}

$FCMIM_{do2a(), do2c()}$ = { before(): callDo2()}

$(M \cup IM \cup AV) \cap FCMIM_{do2a(), do2c()}$ = **{ before(): callDo2()}** // Detected: aspect A1 violates Guideline 7.

Guideline 8: Do minimize a number of aspects which crosscut to the same concern.

Source code:

```
public aspect A2 {
        pointcut callDo2(): call (void source48.C.do2());
        void around(): callDo2(){
                do2b();
                proceed();
        }
        public static void do2b(){
                System.out.println("Do2b");
        }
}
public aspect A1 {

        boolean a = true;

        pointcut callDo2(): call (void source48.C.do2());
        void around(): callDo2(){
                if(a){
                        throw new RuntimeException();
                }else{
                        do2a();
                        proceed();
                }
        }
        public void do2a(){
                System.out.println("Do2a");
        }
}
public class C {
        public void do1(){
                System.out.println("Do1");
```

```
        }
        public void do2(){
                System.out.println("Do2");
        }
        public void do3(){
                System.out.println("Do3");
        }
}
public class Main {

        public static void main(String[] args) {
                C c = new C();
                c.do1();
                c.do2();
                c.do3();
        }

}
```

Violation check expressions:

Consider a unit U.

Let P be the set of pointcuts of U.

Let J_p be the set of corresponding join points of a pointcut p where $p \in P$.

Let PO be the set of pointcuts of other aspect units except U.

Let J_{po} be the set of corresponding join points of a pointcut po where $po \in PO$.

This principle is violated if

$$\exists p \exists po \, (J_p \cap J_{po} \neq \emptyset)$$

Measurement example:

Aspect A1

P = { callDo2()}

$J_{callDo2()}$ = { c.do2();}

PO = { callDo2()}

$J_{callDo2()}$ = { c.do2();}

$J_{callDo2(} \cap J_{callDo2()}$ = { c.do2();} // Detected: aspect A1 violates Guideline 8.

Guideline 9: Do keep scattering behaviors that represent the same concern in one set of corresponding aspects.

Source code:

```
public aspect A1 {
        pointcut callDo2(): call (void source38.C1.do2());
        after(): callDo2(){
                C2.printBlank();
        }
}
public class C1 {
        public void do1(){
                System.out.println("Do1");
        }
        public void do2(){
                System.out.println("Do2");
        }
        public void do3(){
                System.out.println("Do3");
        }
}
public class C2 {
        public void do4(){
                System.out.println("Do4");
        }
        public static void printBlank(){
                System.out.println("");
        }
        public void do5(){
                System.out.println("Do5");
        }
}
public class Main {

        public static void main(String[] args) {
                C1 c1 = new C1();
                c1.do1();
                c1.do2();
                c1.do3();
```

```
        C2 c2 = new C2();
        c2.do4();
        c2.do5();
    }
}
```

Violation check expressions:

Consider a unit U.

Let *M* be the set of methods of U.

Let UCM_m be the set of units calling to a method *m* where $m \in M$.

Let *AC* be the set of all classes in the system.

Let *AAC* be the set of all abstract classes in the system.

Let *PR* be the set of parent units of U.

Let *CD* be the set of children units of U.

This principle is violated if

$$\exists m \, (|UCM_m - AC - AAC - PR - CD - \{U\}| = 1)$$

Additional Condition: Enclosing aspect calling to its enclosed class are not included in $UCMT_m$.

Measurement example:

Class C2

M = { do4(), printBlank(), do5()}

AC = {C1, C2, Main}

AAC = {}

PR = {}

CD = {}

$UCM_{do4()}$ = {Main}

$UCM_{do5()}$ = {Main}

$UCM_{printBlank()}$ = {A1}

$|UCM_{do4()} - AC - AAC - PR - CD - \{C2\}| = 0$

$|UCM_{do5()} - AC - AAC - PR - CD - \{C2\}| = 0$

$|UCM_{printBlank()} - AC - AAC - PR - CD - \{C2\}| = 1$ // Detected: class C2 violates Guideline 9.

Guideline 10: Don't let a base class be dependence on its crosscutting aspects.

Source code:

```
public class C {
        public void do1(){
                System.out.println("Do1");
        }
        public void do2(){
                A1.aspectOf().do2b();
                System.out.println("Do2");
        }
        public void do3(){
                System.out.println("Do3");
        }
}
public aspect A1 {

        pointcut callDo2(): call (void source49.C.do2());
        before(): callDo2(){
                        do2a();
        }
        public void do2a(){
                System.out.println("Do2a");
        }
        public void do2b(){
                System.out.println("Do2b");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *CU* be the set of crosscutting units which crosscuts U.

Let *M* be the set of methods of *CU*.

Let *IM* be the set of inter-type methods of *CU*.

Let UCM_m be the set of units calling to a method *m* where $m \in M$.

Let $UCIM_{im}$ be the set of units calling to an inter-type method *im* where $im \in IM$.

This principle is violated if

$$\exists m\ (U \in (UCM_m)) \vee \exists im\ (U \in (UCIM_{im}))$$

Measurement example:

Class C

$CU = \{A1\}$

$M = \{$ do2a(),do2b()$\}$

$IM = \{\}$

$UCM_{do2a(),\ do2b()} = \{C\}$

$\{C\} \in UCM_{do2a(),\ do2b()}$ // Detected: class C violates Guideline 10.

Guideline 11: Don't use an inheritance relationship to model a crosscutting relationship because an aspect is not a special type of the class that it crosscuts.

Source code:

```
public aspect SavingEnergy extends Home{
   after(): call(static void Home.setUsedWatts(..)) && (!within(SavingEnergy)){
      Home.setUsedWatts(50);
   }
}
public class Home {

        private static int usedWatts = 0;
        protected static int getUsedWatts(){
                return usedWatts;
        }
        protected static void setUsedWatts(int i){
                usedWatts = i;
        }
        protected static void showUsedWatts(){
                System.out.println("This home consumes energy " + usedWatts + " watts");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *PR* be the set of parent units of U.

Let *AC* be the set of all classes in the system.

Let *AAC* be the set of all abstract classes in the system.

This principle is violated if

$$PR \subseteq (AC \cup AAC)$$

Measurement example:

Aspect SavingEnergy

AC = {Home}

AAC = {}

PR = {Home}

$PR \subseteq (AC \cup AAC)$ // Detected: aspect SavingEnergy violates Guideline 11.

Guideline 12: Don't introduce methods to all subclasses to override a concrete method of their abstract superclass.

Source code:

```
public abstract class AC {

}
public class C1 extends AC {

}
public class C2 extends AC {

}
public aspect A1 {
        public void AC.greet(){
                System.out.println("");
        }
        public void C1.greet(){
                System.out.println("Hi");
        }
```

```
        public void C2.greet(){
                System.out.println("Hello");
        }

}
```

Violation check expressions:

Consider a unit U.

Let *SIM* be the set of the similar signature inter-type methods of U.

Let *SU* be the set of sibling units of the same abstract superclass which all sibling units are crosscut by *sim* where $sim \in SIM$.

Let IHM_{su} be the set of concrete methods which *su* are inherited from its abstract superclass where $su \in SU$.

This principle is violated if

$$\exists\, sim\, \forall\, su\ (sim \in IHM_{su})$$

Measurement example:

Aspect A1

SIM = { C1.greet(),C2.greet()}

IHM_{su} = { AC.greet()}

C1.greet() $\in$ IHM_{su}

C2.greet() $\in$ IHM_{su} // Detected: aspect A1 violates Guideline 12.

Guideline 13: Don't define methods of all subaspects to override a concrete method of their abstract aspect.

Source code:

```
public abstract aspect AA {
        abstract pointcut callDo2();
        after(): callDo2(){
                greet();
        }
        protected void greet(){
                System.out.println("");
```

```
        }

}
public aspect A1 extends AA{
        pointcut callDo2(): call (void source51.C.do2());

        protected void greet(){
                System.out.println("I'm Aspect");
        }

}
```

Violation check expressions:

Consider a unit U.

Let *IHM* be the set of concrete methods of U which are inherited from its abstract aspect. Let *M* be the set of methods of U.

Let *SU* be the set of all sibling units of the same abstract aspect of U.

Let M_{su} be the set of methods of *su* where $su \in SU$.

This principle is violated if

$$\forall\, su\ ((IHM \cap M) \cap M_{su} \neq \emptyset)$$

Measurement example:

Aspect A1

M = {greet()}

IHM = {greet()}

(*IHM* ∩ *M*) = {greet()} // Detected: aspect A1 violates Guideline 13.

Guideline 14: Do extract an abstract aspect if the same crosscutting concern has two or more variations.

Source code:

```
public abstract aspect ObserverProtocol {

  protected interface Subject { }
```

```
    protected interface Observer { }
    private WeakHashMap perSubjectObservers;

        protected List getObservers(Subject subject) {
      if (perSubjectObservers == null) {
        perSubjectObservers = new WeakHashMap();
      }
      List observers = (List)perSubjectObservers.get(subject);
      if ( observers == null ) {
        observers = new LinkedList();
        perSubjectObservers.put(subject, observers);
      }
      return observers;
    }

        public void  addObserver(Subject subject, Observer observer) {
      getObservers(subject).add(observer);
    }

    public void removeObserver(Subject subject, Observer observer) {
      getObservers(subject).remove(observer);
    }

    protected abstract pointcut subjectChange(Subject s);

    after(Subject subject): subjectChange(subject) {
      Iterator iter = getObservers(subject).iterator();
      while ( iter.hasNext() ) {
        updateObserver(subject, ((Observer)iter.next()));
      }
    }

    protected abstract void updateObserver(Subject subject, Observer observer);
}
public aspect ColorObserver extends ObserverProtocol{

        declare parents: Point  implements Subject;
        declare parents: Screen implements Observer;

        protected pointcut subjectChange(Subject subject):
                call(void Point.setColor(Color)) && target(subject);
```

```
        protected void updateObserver(Subject subject, Observer observer) {
                ((Screen)observer).display("Screen updated "+
                        "(point subject changed color).");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *CD* be the set of children units of U.

This principle is violated if

$$|CD| \leq 1$$

Measurement example:

Abstract Aspect ObserverProtocol

CD = {ColorObserver}

|*CD*| =1 // Detected: abstract aspect ObserverProtocol violates Guideline 14.

Guideline 15: Don't introduce a method which has the same signature as an inherited method to the realized interface.

Source code:

```
public class SuperC {
    public void methodA() {
        System.out.println("SuperC");
    }
}
public class C extends SuperC{
    public static void main(String[] args) {
        C c = new C();
        c.methodA();
    }
}
public aspect AspectA {
    declare parents: C implements I;
    protected interface I { }
```

```
/* Compilation error: Inter-type declaration conflicts with existing members*/
public void I.methodA() {
      System.out.println("I");
   }*/
}
```

Violation check expressions:

Consider a unit U.

Let *CIM* be the set of concrete inter-type methods of U.

Let *I* be the set of interfaces which are crosscut by *cim* where $cim \in CIM$.

Let *IU* be the set of units implementing interface *i* where $i \in I$.

Let IHM_{iu} be the set of concrete methods which *iu* are inherited from its super-units where $iu \in IU$.

This principle is violated if

$$\exists iu\ (CIM \cap IHM_{iu} \neq \emptyset)$$

Measurement example:

Aspect AspectA

CIM = { I.methodA()}

IU = {C}

IHM_C = { methodA()}

$CIM \cap IHM_{iu}$ = **{ methodA()}** // Detected: aspect AspectA violates Guideline 15.

Heuristic 2.1: All data should be hidden within its class.

Source code:

```
class File {
        private int cylinder = 1;
        private String file_buffer = "Data";
        private boolean error_status = false;
        public int byte_offset = 3; // accidental public data

        public void fopen(){
```

```
            System.out.println("do fopen()" + cylinder);
        }
        public void fclose(){
            System.out.println("do fclose()" + error_status);
        }
        public void fgets(){
            System.out.println("do fgets()" + byte_offset);
        }
        public void fputs(){
            System.out.println("do fputs()" + file_buffer);
        }
}
```

Violation check expressions:

Consider a unit U.

Let *F* be the set of fields of U.

Let *IF* be the set of inter-type fields of U.

Let *PVM* be the set of private members of U.

This principle is violated if

$$(F \cup IF) - PVM \neq \emptyset$$

Measurement example:

Class File

F = { cylinder, file_buffer, error_status, byte_offset }

IF = {}

PVM = { cylinder, file_buffer, error_status }

$(F \cup IF) - PVM$ = **{byte_offset}** // Detected: class File violates Heuristic 2.1.

Heuristic 2.3: Minimize the number of messages in the protocol of a class.

Source code:

```
class LinkedList {

    public void method400(){// inner used
        System.out.println("method400()");
```

```
    }
    public void method800(){// inner used
            System.out.println("method800()");
    }
    public void method1200(){//unused method
            System.out.println("method1200()");
    }
    public void method1600(){//unused method
            System.out.println("method1600()");
    }
    public void method2000(){
            System.out.println("method2000()");
    }
    public void method2400(){
            System.out.println("method2400()");
    }
    public void method2800(){
            System.out.println("method2800()");
    }
    public void method3200(){
            System.out.println("method3200()");
    }
    public void method3600(){
            System.out.println("method3600()");
    }
    public void method4000(){
            System.out.println("method4000()");
    }
}
```

Violation check expressions:

Consider a unit U.

Let *M* be the set of methods of U.

Let *IM* be the set of inter-type methods of U.

Let *DPU* be the set of methods and inter-type methods of U which are used by non-subclass units in different packages.

Let *PBM* be the set of public members of U.

This principle is violated if

$$|((M \cup IM) - DPU) \cap PBM| > 7$$

Measurement example:

Class LinkedList

M = {method400(), method800(), method1200(), method1600(), method2000(), method2400(), method2800(), method3200(), method 3600(), method 4000()}

IM = {}

DPU = {}

PBM = {method400(), method800(), method1200(), method1600(), method2000(), method2400(), method2800(), method3200(), method 3600(), method 4000()}

$|((M \cup IM) - DPU) \cap PBM| = 10$ // Detected: class LinkedList violates Heuristic 2.3.

Heuristic 2.5: Do not put implementation details such as common-code private functions into the public interface of the class.

Source code:

```
class X {

        public void f(){ // f() is used by class X only
                System.out.println("do f()");
        }
        public void f1(){
                System.out.println("do f1()");
                f();
                System.out.println("do f1()(continued)");
        }
        public void f2(){
                System.out.println("do f2()");
                f();
                System.out.println("do f2()(continued)");
        }
}
public class Main {
```

```
    public static void main(String[] args) {
        X x = new X();
        x.f1();
        System.out.println();
        x.f2();
    }

}
```

Violation check expressions:

Consider a unit U.

Let *M* be the set of concrete methods of U.

Let *IM* be the set of concrete inter-type methods of U.

Let $FCMIM_{mim}$ be the set of functions (methods/ inter-type methods/ advices) in other units calling to a method or an inter-type methods *mim* (except the nested classes of U and descendant classes of U) where $mim \in (M \cup IM)$.

Let *PBM* be the set of public members of U.

The principle is violated if

$$\exists\, mim\ [\ (FCMIM_{mim} = \emptyset) \wedge (mim \in PBM)]$$

Measurement example:

Class X

M = {f(), f1(), f2()}

$FCMIM_{f()}$ = {}

$FCMIM_{f1()}$ = {Main}

$FCMIM_{f2()}$ = {Main}

PBM = {f(), f1(), f2()}

$((FCMIM_{f()} = \emptyset) \wedge (f() \in PBM)$ // Detected: class X violates Heuristic 2.5.

Heuristic 2.6: Do not clutter the public interface of a class with items that users of that class are not able to use or are not interested in using it.

Source code:

```
public abstract class SuperX {
        public SuperX(){// public constructor but only called by its subclass
                System.out.println("Constructor of SuperX");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *C* be the set of constructors of U.

Let *PBM* be the set of public members of U.

This principle is violated if

$$C \cap PBM \neq \emptyset$$

Measurement example:

Abstract Class SuperX

C = { SuperX()}

PBM = { SuperX()}

$C \cap PBM$ = **{ SuperX()}** // Detected: abstract class SuperX violates Heuristic 2.6.

Heuristic 2.7: Classes should only exhibit nil or export coupling with other classes, that is, a class should only use operations in the public interface of another class or have nothing to do with that class.

Source code:

```
class File {
        public int byte_offset;

        public int getByte_offset(){
                return byte_offset;
        }
        public void setByte_offset(int i){
                byte_offset = i;
        }
```

```
        public void print(){
                System.out.println("Byte_offset = " + byte_offset);
        }
}
public class Main {
        public static void main(String[] args) {
                File f = new File();
                f.byte_offset = 1;// direct acess field of other class
                f.print();
        }
}
```

Violation check expressions:

Consider a unit U.

Let *FT* be the set of functions (methods/ intertype-methods/ advices) of U.

Let $NPBMBOU_{ft}$ be the set of members (fields/ inter-type fields/ non-public methods/ non-public inter-type methods/ non-public pointcuts) of other units except U which are used by *ft* where $ft \in FT$.

The principle is violated if

$$\exists ft[\ NPBMBOU_{ft} \neq \emptyset]$$

Measurement example:

Class Main

FT = {main()}

$NPBMBOU_{main()}$ = **{byte_offset}** // Detected: class Main violates Heuristic 2.7.

Heuristic 2.9: Keep related data and behavior in one place.

Source code:

```
class Oven {
        private boolean gaslevel = true;
        private int actualtemp = 100;
        private int desiredtemp = 100;
        private boolean valvestatus = true;
        private boolean pilotlightstatus = false;
```

```
        public boolean getGaslevel(){
                return gaslevel;
        }
        public int getActualtemp(){
                return actualtemp;
        }
        public int getDesiredlevel(){
                return desiredtemp;
        }
        public boolean getValvestatus(){
                return valvestatus;
        }
        public boolean getPilotlightstatus(){
                return pilotlightstatus;
        }
}

public class User {

        public static boolean check_preheated(Oven o){
                if ((o.getActualtemp() >= o.getDesiredlevel())
                                && o.getValvestatus() && o.getGaslevel()){
                        return true; // oven is heated.
                }
                return false;
        }

        public static void main(String[] args) {
                Oven o = new Oven();
                if(check_preheated(o)){
                        System.out.println("The oven is heated");
                }else
                        System.out.println("The oven is not heated");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *GM* be the set of get methods of U.

Let *GIM* be the set of get inter-type methods of U.

This principle is violated if

$$|GM \cup GIM| > 4$$

Measurement example:

Class Oven

GM = { getGaslevel(), getActualtemp(),getDesiredlevel(),getValvestatus(), getPilotlightstatus()}

GIM = {}

$|GM \cup GIM| = 5$ // Detected: class Oven violates Heuristic 2.9.

Heuristic 2.10: Spin off non-related information into another class (i.e., non-communicating behavior).

Source code:

```
class VagueClass {

        private String data1 = "data1";
        private String data2 = "data2";

        public void f1(){ // use data1
                System.out.println("do f1()uses " + data1);
        }
        public void f2(){ // use data1
                System.out.println("do f2()uses " + data1);
        }
        public void f3(){ // use data2
                System.out.println("do f3()uses " + data2);
        }
        public void f4(){ // use data2
                System.out.println("do f4()uses " + data2);
        }
}
```

Violation check expressions:

Consider a unit U.

Let *NSF* be the set of non-static fields of U.

Let *MC* be the set of methods or constructors of U.

Let $MCUF_{nsfx}$ be the set of methods or constructors using nsf_x where $nsf_x \in NSF$.

Let $MCUF_{nsfy}$ be the set of methods or constructors using nsf_y where $nsf_y \in (NSF-\{nsf_x\})$.

Let $MCUMC_{mcufnsfx}$ be the set of methods or constructors using *mcufnsfx* where $mcufnsfx \in MCUMC_{mcufnsfx}$.

Let $MCUMC_{mcufnsfy}$ be the set of methods or constructors using *mcufnsfy* where $mcufnsfy \in MCUMC_{mcufnsfy}$.

This principle is violated if

$$((|NSF|> 1) \wedge (|MC|> 1)) \wedge (\exists nsfx \exists nsfy\ ((MCUF_{nsfx} \neq \emptyset) \wedge (MCUF_{nsfy} \neq \emptyset) \wedge (MCUF_{nsfx} \cap MCUF_{nsfy} = \emptyset)) \wedge (\forall mcufnsfx\ \forall mcufnsfy\ ((mcufnsfy \notin MCUMC_{mcufnsfx}) \wedge (mcufnsfx \notin MCUMC_{mcufnsfy}) \wedge (MCUMC_{mcufnsfx} \cap MCUMC_{mcufnsfy} = \emptyset))$$

Measurement example:

Class VagueClass

NSF = {data1, data2}

MC = {f1(), f2(), f3(), f4()}

$MCUF_{data1}$ = {f1(), f2()}

$MCUF_{data2}$ = {f3(), f4()}

$MCUMC_{mcufdata1}$ = {}

$MCUMC_{mcufdata2}$ = {}

$((|NSF|= 2) \wedge (|MC| = 4)) \wedge ((MCUF_{data1} = \{f1(), f2()\}) \wedge (MCUF_{data2} = \{f3(), f4()\}) \wedge (MCUF_{data1} \cap MCUF_{data2} = \emptyset) \wedge ((f3() \notin MCUMC_{mcufdata1}) \wedge (f4() \notin MCUMC_{mcufdata1}) \wedge (f1() \notin MCUMC_{mcufdata2}) \wedge (f2() \notin MCUMC_{mcufdata2}))$ //

Detected: class VagueClass violates Heuristic 2.10.

Heuristic 3.2: Do not create god classes/ objects in your system. Be very suspicious of a class whose name contains Driver, Manager, System, or Subsystem.

Source code:

```
public class CookingManager {

        private Oven o;
        private Pan p;

        public void startCook(){
                o = new Oven();
                o.setActualtemp(110);
                p = new Pan();
                p.setActualtemp(220);
        }

        public void overheat(){
                if ((o.getActualtemp() > 100)||(p.getActualtemp() > 200)){
                        System.out.println("Oven or pan is overheated");
                        o.setActualtemp(10);
                        p.setActualtemp(20);
                }
        }

}
```

Violation check expressions:

Consider a unit U.

Let *UDMSSS* be the set of units of the system that contain Driver, Manager, System, or SubSystem at the end of their names.

This principle is violated if

$$U \in UDMSSS$$

Measurement example:

Class CookingManager

UDMSSS = {CookingManager}

CookingManager $\in$ *UDMSSS* // Detected: class CookingManager violates Heuristic 3.2.

Heuristic 3.7: Eliminate irrelevant classes from your design.

Source code:

```
class Color {

        private String cName;

        public void setCName(String n){
                cName = n;
        }
        public String getCName(){
                return cName;
        }
        public void printCName(){
                System.out.println("This color is " + cName);
        }
}
```

Violation check expressions:

Consider a unit U.

Let *FT* be the set of functions (methods/constructors) of U.

Let C be the set of constructors of U.

Let *GM* be the set of get methods of U which each get method has only one statement returning a field in get method's name.

Let *SM* be the set of set methods of U which each set method has only one statement setting a field in set method's name.

Let *PTM* be the set of print type methods of U which each print type method has only one statement printing a field in print type method's name.

Let *AEX* be the set of exception classes in the system.

This principle is violated if

$$(FT \neq \emptyset) \wedge (FT - (C \cup GM \cup SM \cup PTM) = \emptyset) \wedge (U \notin AEX)$$

Measurement example:

Class Color

FT = { setCName(String n), getCName(), printCName()}

C = {}

GM = {getCName()}

SM = {setCName(String n) }

PTM = {printCName()}

AEX = {}

(*FT*={ setCName(String n), getCName(), printCName()}) ∧ (*FT* – (*C* ∪ *GM* ∪ *SM* ∪ *PTM*) = Ø) ∧(Color ∉*AEX*) // Detected: class Color violates Heuristic 3.7.

Heuristic 3.8: Eliminate classes that are outside the system.

Source code:

```
class Blender {

        public void whip(){
                System.out.println("do whip");
        }
        public void chop(){
                System.out.println("do chop");
        }
        public void puree(){
                System.out.println("do puree");
        }
        public void liquify(){
                System.out.println("do liquify");
        }
}
public class RegistrationCard {

        LinkedList productList = new LinkedList();

        public void registerProduct(String name1){
                productList.add(name1);
```

```
        }
        public void listProduct(){
                for(int i=0; i < productList.size(); i++){
                        System.out.println(productList.get(i));
                }
        }
}
public class Main {

        public static void main(String[] args) {
                RegistrationCard rc = new RegistrationCard();
                rc.registerProduct("blender");
                rc.registerProduct("toaster");
                rc.registerProduct("blender");
                rc.registerProduct("television");
                rc.listProduct();
        }
}
```

Violation check expressions:

Consider a unit U.

Let *FT* be the set of members (methods/ inter-type methods/ constructors) of U.

Let UCM_{ft} be the set of units calling to a function *ft* where $ft \in FT$.

Let *P* be the set of pointcuts of U.

Let J_p be the set of corresponding join points of a pointcut *p* where $p \in P$.

This principle is violated if

$$(FT \neq \emptyset) \wedge \forall ft\ (UCM_{ft} - \{U\} = \emptyset) \wedge \forall p\ (J_p = \emptyset)$$

Measurement example:

Class Blender

$FT = \{$ whip(), chop(), puree(), liquify()$\}$

$UCM_{whip()} = \{\}$

$UCM_{chop()} = \{\}$

$UCM_{puree()} = \{\}$

$UCM_{liquify()} = \{\}$

FT = { whip(), chop(), puree(), liquify()} $\wedge$ ($UCM_{whip()}$ – {Blender} = Ø) $\wedge$ ($UCM_{chop()}$ – {Blender} = Ø) $\wedge$ ($UCM_{puree()}$ – {Blender} = Ø) $\wedge$ ($UCM_{liquefy()}$ – {Blender} = Ø) // Detected: class Blender violates Heuristic 3.8.

Heuristic 3.10: Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

Source code:

```
class Librarian {

        private BookShelf bs;

        public Librarian(BookShelf bs){
                this.bs = bs;
        }

        public void put_me_on_shelf(Book b){
                bs.put_book_on_shelf(b);
        }

}
```

Violation check expressions:

Consider a unit U.

Let M be the set of methods of U.

Let ST_m be the set of statements of a method m where $m \in M$.

Let MSG_{stm} be the set of messages contained in a statement stm of a method m where $stm \in ST_m$.

Let ADC be the set of adapter classes in the system.

Let PX be the set of proxy classes in the system.

This principle is violated if

$$\forall m((|ST_m| = 1) \wedge \forall st\ (|MSG_{stm}| = 1)) \wedge (U \notin ADC) \wedge (U \notin PX)$$

Measurement example:

Class Librarian

M = { put_me_on_shelf(Book b)}

$ST_{put\_me\_on\_shelf(Book\ b)}$ = { bs.put_book_on_shelf(b)}

$MSG_{bs.put\_book\_on\_shelf(b)}$ = { put_book_on_shelf(b)}

ADC = {}

PX = {}

$(|ST_{put\_me\_on\_shelf(Book\ b)}| = 1) \wedge (|MSG_{bs.put\_book\_on\_shelf(b)}| = 1)) \wedge$ (Librarian $\notin ADC$) $\wedge$ (Librarian $\notin PX$) // Detected: class Librarian violates Heuristic 3.10.

Heuristic 4.5: If a class contains objects of another class, then the containing class should be sending messages to the contained objects, that is, the containment relationship should always imply a use relationship.

Source code:

```
public class Meal {
        Pie p;
        Melon m;

        public Meal(Pie p, Melon m){
                this.p = p;
                this.m = m;
        }
        public void reportMeal(){
                System.out.println("This meal contains Pie and Melon");
        }
}

public class Melon {
        public void coolMelon(){
                System.out.println("This Pie is cooled");
        }
}

public class Pie {
        public void warmPie(){
                System.out.println("This Pie is warmed");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *CT* be the set of class type of field or inter-type field types which are contained by U.

Let *M* be the set of methods of *CT*.

Let *IM* be the set of inter-type methods of *CT*.

Let *F* be the set of fields of *CT*.

Let *IF* be the set of fields of *CT*.

Let $UCMIM_{mim}$ be the set of units calling to a method or an inter-type method *mim* where $mim \in (M \cup IM)$.

Let $UUFIF_{fif}$ be the set of units using a field or an inter-type field *fif* where $fif \in (F \cup IF)$.

This principle is violated if

$$\forall\, mim\ (U \notin UCMIM_{mim}) \wedge \forall\, fif\ (U \notin UUFIF_{fif})$$

Measurement example:

Class Meal

CT = {Pie, Melon}

M = {coolMelon(), warmPie()}

IM = {}

F = {}

IF = {}

$UCMIM_{coolMelon(),\ warmPie()}$ = {}

(Meal $\notin$ $UCMIM_{coolMelon(),\ warmPie()}$) // Detected: class Meal violates Heuristic 4.5.

Heuristic 4.7: Classes should not contain more objects than a developer can fit in his or her short-term memory. A favorite value for this number is six.

Source code:

```
public class Meal {
```

```
        boolean mealStatus = true;
        private int melon = 1;
        private int steak = 2;
        private int potato = 3;
        private int peas = 4;
        private int corn = 5;
        private int pie = 6;

        public void reportMeal(){
                if(mealStatus){
                        System.out.println("This meal contains melon = "+ melon);
                        System.out.println("This meal contains steak = "+ steak);
                        System.out.println("This meal contains melon = "+ potato);
                        System.out.println("This meal contains peas = "+ peas);
                        System.out.println("This meal contains corn = "+ corn);
                        System.out.println("This meal contains pie = "+ pie);
                }
        }
}
```

Violation check expressions:

Consider a unit U.

Let *F* be the set of fields of U.

Let *IF* be the set of fields of U.

This principle is violated if

$$|F \cup IF| > 6$$

Measurement example:

Class Meal

F = {mealStatus, melon, steak, potato, peas, corn, pie}

IF = {}

$|F \cup IF| = 7$ // Detected: class Meal violates Heuristic 4.7.

Heuristics 4.13: A class must know what it contains, but it should never know who contains it.

Source code:

```
public class AlarmClock {

        BedRoom br;
        String type = "Digital AlarmClock";
        public AlarmClock(BedRoom br){
                this.br = br;
                br.ownerBedRoom() ;
        }
        public String getType(){
                return type;
        }
}

public class BedRoom {

        AlarmClock ac;
        public BedRoom(){
                ac = new AlarmClock(this);
        }
        public void showBedRoom(){
                System.out.println("This BedRoom contains " + ac.getType());
        }
        public void ownerBedRoom(){
                System.out.println("Own by Sunnie");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *UC* be the set of units which contain U type fields or U type inter-type fields.

Let *M* be the set of methods of *UC*. Let *IM* be the set of inter-type methods of *UC*.

Let *F* be the set of fields of *UC*. Let *IF* be the set of fields of *UC*.

Let $UCMIM_{mim}$ be the set of units calling to a method or an inter-type method *mim* where $mim \in (M \cup IM)$.

Let $UUFIF_{fif}$ be the set of units using a field or an inter-type field *fif* where $fif \in (F \cup IF)$.

This principle is violated if

$$\exists\, mim\ (U \in UCMIM_{mim}) \vee \exists\, fif\ (U \in UUFIF_{fif})$$

Measurement example:

Class AlarmClock

UC = {BedRoom}

M = { showBedRoom(), ownerBedRoom()}

IM = {}

F = {ac}

IF = {}

$UCMIM_{showBedRoom(),\ ownerBedRoom()}$ = {Main, AlarmClock}

$UUFIF_{ac}$ = {Main}

AlarmClock $\in$ $UCMIM_{showBedRoom(),\ ownerBedRoom()}$ // Detected: class AlarmClock violates Heuristic 4.13.

Heuristics 4.14: Objects that share lexical scope-those contained in the same containing class should not have uses relationships between them.

Source code:

```
public class ATM {
        Keypad k;
        CardReader cr;
        Display d;

        public ATM(Keypad k, CardReader cr, Display d){
                this.k = k;
                this.cr = cr;
                this.d = d;
        }
        public void useATM(){
                cr.have_a_card(d, k);
        }
```

```
}

public class CardReader {

        public void have_a_card(Display d, Keypad k){
                System.out.println("do Have_a_card?");
                k.get_pin();
                d.display_pin();

        }
}

public class Display {
        public void display_pin(){
                System.out.println("do display_pin");
        }
}

public class Keypad {
        public void get_pin(){
                System.out.println("do get_pin");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *UC* be the set of units which contain U type fields or U type inter-type fields.

Let *CT* be the set of class type of field or inter-type field types which are also contained by *UC*.

Let *M* be the set of methods of *CT*.

Let *IM* be the set of inter-type methods of *CT*.

Let *F* be the set of fields of *CT*.

Let *IF* be the set of fields of *CT*.

Let $UCMIM_{mim}$ be the set of units calling to a method or an inter-type method *mim* where $mim \in (M \cup IM)$.

Let $UUFIF_{fif}$ be the set of units using a field or an inter-type field *fif* where $fif \in (F \cup IF)$.

This principle is violated if

$$\exists mim\ (U \in UCMIM_{mim}) \vee \exists fif\ (U \in UUFIF_{fif})$$

Measurement example:

Class CardReader

UC = {ATM}

CT = {Display, KeyPad}

M = { display_pin(), get_pin()}

$UCMIM_{display\_pin(),\ get\_pin()}$ = {Main, CardReader}

CardReader $\in$ $UCMIM_{display\_pin(),\ get\_pin()}$ // Detected: class CardReader violates Heuristic 4.14.

Heuristic 5.2: Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.

Source code:

```
public class Animal {
        public void animalSound(){
                if (this instanceof Dog){
                        System.out.println("Wolf");
                }else if(this instanceof Cat){
                        System.out.println("Meaw");
                }else
                        System.out.println("...");
        }
}
public class Dog extends Animal{

}
public class Cat extends Animal{

}
```

Violation check expressions:

Consider a unit U.

Let *FT* be the set of functions (methods/ inter-type methods/ advices) of U.

Let $TPISF_{ft}$ be the set of types defined in predicates of if statement of a function *ft* where $ft \in FT$.

Let *CD* be the set of children units of U.

This principle is violated if

$$\exists ft\ (CD \cap TPISF_{ft} \neq \emptyset) \vee (U \in TPISF_{ft})$$

Measurement example:

Class Animal

FT = {animalSound()}

CD = {Dog, Cat}

$TPISF_{animalSound()}$= {Animal}

$(CD \cap TPISF_{animalSound()} = \emptyset) \vee (\text{Animal} \in TPISF_{animalSound()})$ // Detected: class Animal violates Heuristic 5.2.

Heuristic 5.5: In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six.

Source code:

```
public class Fruit {

}
public class TropicalFruit extends Fruit{
        public TropicalFruit(){
                super();
                System.out.println("TropicalFruit extends Fruit");
        }
}
public class GreenTropicalFruit extends TropicalFruit {
        public GreenTropicalFruit(){
                super();
                System.out.println("GreenTropicalFruit extends TropicalFruit");
        }
}
public class GreenTropicalFruitFromPacific extends GreenTropicalFruit {
```

```
        public GreenTropicalFruitFromPacific(){
                super();
                System.out.println("GreenTropicalFruitFromPacific extends GreenTropicalFruit");
        }
}
public class GreenTropicalFruitFromAsiaPacific extends GreenTropicalFruitFromPacific {
        public GreenTropicalFruitFromAsiaPacific(){
                super();
                System.out.println("GreenTropicalFruitFromAsiaPacific extends GreenTropicalFruitFromPacific");
        }
}
public class GreenTropicalFruitFromThailand extends GreenTropicalFruitFromAsiaPacific {
        public GreenTropicalFruitFromThailand(){
                super();
                System.out.println("GreenTropicalFruitFromThailand extends GreenTropicalFruitFromAsiaPacific");
        }
}
public class GreenTropicalFruitFromChiangMai extends GreenTropicalFruitFromThailand {
        public GreenTropicalFruitFromChiangMai(){
                super();
                System.out.println("GreenTropicalFruitFromChiangMai extends GreenTropicalFruitFromThailand");

        }
}
public class Kiwi extends GreenTropicalFruitFromChiangMai {
        public Kiwi(){
                super();
                System.out.println("Kiwi extends GreenTropicalFruitFromChiangMai");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *dit(U)* be the function to determine depth of inheritance of U.

This principle is violated if

$$dit(U) > 6$$

Measurement example:

Class Kiwi

*dit(*Kiwi*)* = 7 // Detected: class Kiwi violates Heuristic 5.5.

Heuristic 5.6: All abstract classes must be base classes.

Source code:

```
public abstract class AllEmployees {
        int salary;
        int sicktime;
        int MedicationPlan;
        public void taxes(){
                System.out.println("do taxes");
        }
        public void benefits(){
                System.out.println("do benefits");
        }
}
public class Main {

        public static void main(String[] args) {
                System.out.println("do main");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *RU* be the set of realization units of U.

Let *CD* be the set of children units of U.

This principle is violated if

$$RU \cup CD = \emptyset$$

Measurement example:

Abstract Class AllEmployees

RU = {}

CD = {}

$RU \cup CD = \emptyset$ // Detected: abstract class AllEmployees violates Heuristic 5.6.

Heuristics 5.9: If two or more classes share only common data (no common behavior), then that common data should be placed in a class that will be contained by each sharing class.

Source code:

```
public class Fruit {
}
public class Orange extends Fruit {
        private int weight;
        private String color;
        public Orange(){
                weight = 2;
                color = "orange";
        }
        public void showOrange(){
                System.out.println("Orange property: " + weight + " , " + color);
        }

}
public class Apple extends Fruit {
        private int weight;
        private String color;
        public Apple(){
                weight = 1;
                color = "red";
        }
        public void showApple(){
                System.out.println("Apple property: " + weight + " , " + color);
        }
}
```

Violation check expressions:

Consider a unit U.

Let *SU* be the set of sibling units of U.

Let *F* be the set of fields of U.

Let F_{su} be the set of fields of *su* where $su \in SU$.

This principle is violated if

$$\forall su\ (F \cap F_{su} \neq \emptyset)$$

Measurement example:

Class Apple

SU = {Orange}

F = {weight, color}

F_{Orange} = {weight, color}

$F \cap F_{Orange}$ = **{weight, color}** // Detected: class Apple violates Heuristic 5.9.

Heuristics 5.10: If two or more classes have common data and behavior, then those classes should each inherit from a common base class that captures those data and methods.

Source code:

```
public class Fruit {
}
public class Apple extends Fruit {
        private int weight;
        private String color;
        public Apple(){
                weight = 1;
                color = "red";
        }
        public void showApple(){
                System.out.println("Apple property: " + weight + " , " + color);
        }
        public void taste(){
                System.out.println("Sweet & Sour");
        }
}
public class Orange extends Fruit {
        private int weight;
        private String color;
        public Orange(){
                weight = 2;
```

```
            color = "orange";
        }
        public void showOrange(){
            System.out.println("Orange property: " + weight + " , " + color);
        }
        public void taste(){
            System.out.println("Sweet & Sour");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *SU* be the set of sibling units of U.

Let *F* be the set of fields of U.

Let F_{su} be the set of fields of *su* where $su \in SU$.

Let *M* be the set of methods of U.

Let M_{su} be the set of methods of *su* where $su \in SU$.

Let ST_m be the set of statements in a method *m* where $m \in (M \cap M_{su})$.

Let ST_{msu} be the set of statements in a method *msu* where $msu \in (M_{su} \cap M)$.

This principle is violated if

$$(\forall su\ (F \cap F_{su} \neq \emptyset) \wedge (M \cap M_{su} \neq \emptyset) \wedge (\forall m \forall msu\ (ST_m \subseteq ST_{msu} \wedge ST_{msu} \subseteq ST_m))$$

Measurement example:

Class Orange

SU = {Apple}

F = {weight, color}

F_{Apple} = {weight, color}

M = {showOrange(), taste()}

M_{Apple} = {showApple(), taste()}

$ST_{showOrange()}$ = { System.out.println("Orange property: " + weight + " , " + color);}

$ST_{Orange.taste()}$ = { System.out.println("Sweet & Sour");}

$ST_{showApple()}$ = { System.out.println("Apple property: " + weight + " , " + color);}

$ST_{Apple.taste()}$ = { System.out.println("Sweet & Sour");}

$F \cap F_{Apple}$ = {weight, color} $\wedge$ ($M \cap M_{Apple}$ = {taste()}) $\wedge (ST_{Orange.taste()} \subseteq ST_{Apple.taste()} \wedge ST_{Apple.taste()} \subseteq ST_{Orange.taste()})$ // Detected: class Orange violates Heuristic 5.10.

Heuristics 5.13: Explicit case analysis on the value of an attribute is often an error. The class should be decomposed into an inheritance hierarchy, where each value of the attribute is transformed into a derived class.

Source code:

```
public class Ball {

        protected int bounceFactor;

        public void doBehavior(){
                switch (bounceFactor) {
                case 1: System.out.println("Hi! I'm a red ball"); break;
                case 2:    System.out.println("Hi! I'm a green ball"); break;
                case 3: System.out.println("Hi! I'm a yellow ball"); break;
                }
        }
}
public class RedBall extends Ball{
        public RedBall(){
                bounceFactor = 1;
        }
}
public class YellowBall extends Ball{
        public YellowBall(){
                bounceFactor = 3;
        }
}
public class GreenBall extends Ball{
        public GreenBall(){
                bounceFactor = 2;
        }
}
```

Violation check expressions:

Consider a unit U.

Let *FT* be the set of functions (methods/ inter-type methods/ advices) of U.

Let $SPPTF_{ft}$ be the set of switch statements having primitive type fields in their predicates of a function *ft* where $ft \in FT$.

This principle is violated if

$$\exists ft\ (SPPTF_{ft} \neq \emptyset)$$

Measurement example:

Class Ball

FT = {doBehavior()}

$SPPTF_{doBehavior()}$ = **{ switch (bounceFactor) }** // Detected: class Ball violates Heuristic 5.13.

Heuristics 5.15: Do not turn objects of a class into derived classes of the class. Be very suspicious of any derived class for which there is only one instance.

Source code:

```
public class CarManufacturer {
        protected int totalSales = 100;
}
public class GeneralMotors extends CarManufacturer{

        public void accounting(){
                System.out.println("Profit = " + (totalSales*0.3));
        }
}
public class Chrysler extends CarManufacturer{

        public void accounting(){
                System.out.println("Profit = " + (totalSales*0.5));
        }
}
public class Ford extends CarManufacturer{
```

```
        public void accounting(){
                System.out.println("Profit = " + (totalSales*0.4));
        }
}
public class Main {

        public static void main(String[] args) {
                Chrysler c = new Chrysler();
                c.accounting();
                Ford f = new Ford();
                f.accounting();
                GeneralMotors gm = new GeneralMotors();
                gm.accounting();
        }

}
```

Violation check expressions:

Consider a unit U.

Let *NAPR* be the set of non-abstract parent units of U except class libraries.

Let *IS* be the set of instance of U.

Let *STC* be the set of singleton classes in the system.

This principle is violated if

$$(NAPR \neq \emptyset) \wedge (|IS| \leq 1) \wedge (U \notin STC)$$

Measurement example:

Class Ford

NAPR = {CarManufacturer}

IS = {new Ford()}

STC = {}

(*NAPR* = {CarManufacturer}) $\wedge$ $(|IS| = 1) \wedge$ **(Ford** $\notin$ ***STC*)** // Detected: class Ford violates Heuristic 5.15.

Heuristics 5.17: It should be illegal for a derived class to override a base class method with a NOP method, that is, a method that does nothing.

Source code:

```
public class Stack {
        public void pop(){
                System.out.println("do pop");
        }
}
public class EmptyStack extends Stack {
        public void pop(){

        }
}
```

Violation check expressions:

Consider a unit U.

Let *OVM* be the set of methods of U which override inherited methods.

Let $ihm \in IHM$.

Let ST_{ovm} be the set of statements in a method *ovm* where $ovm \in OVM$.

This principle is violated if

$$\exists\, ovm\ (ST_{ovm} = \emptyset)$$

Measurement example:

Class EmptyStack

$OVM = \{pop()\}$

$ST_{pop()} = \{\}$ // Detected: class EmptyStack violates Heuristic 5.17.

Heuristics 9.2: Do not change the state of an object without going through its public interface.

Source code:

```
public class AirTrafficController {
        public void newPlaneIn(Plane p){
                AirTrafficMemory.addPlane(p);
                AirTrafficMemory.noOfAirplane++;
        }
```

```
        static class AirTrafficMemory{
                static LinkedList airplane = new LinkedList();
                static int noOfAirplane = 0;
                static void addPlane(Plane p){
                        if(!airplane.contains(p)){
                                airplane.add(p);
                        }
                }
                static void showPlanes(){
                        System.out.println(noOfAirplane+" airplanes contain:");
                        for(int i=0; i<airplane.size(); i++){
                                System.out.println(airplane.get(i));
                        }
                }

        }
}
public class Plane {

}
```

Violation check expressions:

Consider a unit U.

Let *FO* be the set of fields of other units except U.

Let *IFO* be the set of inter-type fields of other units except U.

Let UWF_{fo} be the set of units writing to a field *fo* where $fo \in FO$.

Let $UWIF_{ifo}$ be the set of units writing to an inter-type field *ifo* where $ifo \in IFO$.

This principle is violated if

$$\exists\, fo\ (U \in UWF_{fo}) \vee \exists\, ifo\ (U \in UWIF_{ifo})$$

Measurement example:

Class AirTrafficController

FO = {airplane, noOfAirplane}

IFO = {}

$UWF_{airplane,\ noOfAirplane}$ = {AirTrafficController}

AirTrafficController $\in$ $UWF_{airplane,\ noOfAirplane}$ // Detected: class AirTrafficController violates Heuristic 9.2.

Duplicate Code: The same code structure in two or more places is a good sign that the code needs to be refactored: if you need to make a change in one place, you'll probably need to change the other one as well, but you might miss it.

Source code:

```
public class C1 {
        public void do1(){
                System.out.println("Do 1");
                doPrint();
        }
        public void doPrint1(){
                System.out.println("Print1");
                System.out.println("Print2");
                System.out.println("Print3");
        }
}
public class C2 {
        public void do2(){
                System.out.println("Do 2");
                doPrint();
        }
        public void doPrint2(){
                System.out.println("Print1");
                System.out.println("Print2");
                System.out.println("Print3");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *M* be the set of methods of U.

Let *OM* be the set of other methods of all units except the given method *m* where $m \in M$.

Let ASM_m be the set of adjacent statements of *m*.

Let ASM_{om} be the set of adjacent statements of *om* where $om \in OM$.

The principle is violated if

$$\exists m \exists om \ (ASM_m \cap ASM_{om} \neq \emptyset)$$

Measurement example:

Class C1

$ASM_{do1()}$ = { ØSystem.out.println("Do 1");, System.out.println("Do 1"); doPrint();, doPrint();Ø }

$ASM_{doPrint1()}$ = { ØSystem.out.println("Print1");, System.out.println("Print1"); System.out.println("Print2");, System.out.println("Print2"); System.out.println("Print3");, System.out.println("Print3");Ø }

$ASM_{do2()}$ = { ØSystem.out.println("Do 2");, System.out.println("Do 2"); doPrint();, doPrint();Ø }

$ASM_{doPrint2()}$ = { ØSystem.out.println("Print1");, System.out.println("Print1"); System.out.println("Print2");, System.out.println("Print2"); System.out.println("Print3");, System.out.println("Print3");Ø }

$ASM_{do1()} \cap ASM_{doPrint1()} = \{\}$

$ASM_{do1()} \cap ASM_{do2()} = \{\}$

$ASM_{do1()} \cap ASM_{doPrint2()} = \{\}$

$ASM_{doPrint1()} \cap ASM_{do2()} = \{\}$

$ASM_{doPrint1()} \cap ASM_{doPrint2()}$ = { ØSystem.out.println("Print1");, System.out.println("Print1"); System.out.println("Print2");, System.out.println("Print2"); System.out.println("Print3");, System.out.println("Print3");Ø } // Detected: class C1 violates Duplicate Code.

Long Method: Systems that have a majority of small methods tend to be easier to extend and maintain because they're easier to understand and contain less duplication.

Source code:

```
public class C {

    public void do1(){
```

```
            System.out.println("**********");
            System.out.println("----------");
            System.out.println("Do 1.1");
            System.out.println("Do 1.2");
            System.out.println("Do 1.3");
            System.out.println("Do 1.4");
            System.out.println("Do 1.5");
            System.out.println("Do 1.6");
            System.out.println("Do 1.7");
            System.out.println("Do 1.8");
            System.out.println("Do 1.9");
            System.out.println("Do 1.10");
            System.out.println("**********");
            System.out.println("----------");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *FT* be the set of functions (methods/ inter-type methods/ advices) of U.

Let ST_{ft} be the set of statements of a function *ft* where $ft \in FT$.

This unit is violated if

$$\exists ft\ (|ST_{ft}| > 13)$$

Measurement example:

Class C

FT = {do1()}

$ST_{do1()}$ = { System.out.println("**********");, System.out.println("----------");, System.out.println("Do 1.1");, System.out.println("Do 1.2");, System.out.println("Do 1.3");, System.out.println("Do 1.4");, System.out.println("Do 1.5");, System.out.println("Do 1.6");, System.out.println("Do 1.7");, System.out.println("Do 1.8");, System.out.println("Do 1.9");, System.out.println("Do 1.10");, System.out.println("**********");, System.out.println("----------");}

$|ST_{do1()}| = 14$ // Detected: class C violates Long Method.

Long Parameter List: Long parameter lists are hard to understand. You don't need to pass in everything a method needs, just enough so it can find all it needs.

Source code:

```
public class C1 {
        public void greeting(String name, String greet){
                System.out.println(greet + " Ms. " + name);
        }
}
public class C2 {
        private String s = "Hello";

        public String getS(){
                return s;
        }
} public class Main {

        public static void main(String[] args) {
                C1 c1 = new C1();
                C2 c2 = new C2();
                c1.greeting("Sunnie", c2.getS());
        }
}
```

Violation check expressions:

Consider a unit U.

Let *M* be the set of methods of U.

Let *IM* be the set of inter-type methods of U.

Let $UGDFP_{mim}$ be the set of units getting other's data and sending them (without any pre-processing) to *mim* where $mim \in (M \cup IM)$.

This principle is violated if

$$\exists\, mim\ UGDFP_{mim} \neq \emptyset$$

Measurement example:

Class C1

M = {greeting()}

$UGDFP_{greeting()}$ = {Main} // Detected: class C1 violates Long Parameter List.

Feature Envy: Data and behavior that acts on that data belong together. When a method makes too many calls to other classes to obtain data or functionality, Feature Envy is in the air.

Source code:

```
public class C1 {
        public void greeting(String name){
                System.out.println(C2.getS() + " Ms. " + name);
        }
}
public class C2 {
        private String s = "Hello";

        public String getS(){
                return s;
        }
        public static void main(String[] args) {
                C1 c1 = new C1();
                c1.greeting("Sunnie");
        }
}
```

Violation check expressions:

Consider a unit U.

Let *F* be the set of fields of U.

Let *IF* be the set of fields of U.

Let *M* be the set of methods of U.

Let *IM* be the set of inter-type methods of U.

Let *STM* be the set of static methods of U.

Let $FSMTU_{stm}$ be the set of fields that the static method *stm* uses where $stm \in STM$.

Let $IFSMTU_{stm}$ be the set of inter-type fields that the static method *stm* uses where $stm \in STM$.

Let $MSMTU_{stm}$ be the set of methods that the static method *stm* uses where $stm \in STM$.

Let $IMSMTU_{stm}$ be the set of inter-type methods that the static method *stm* uses where $stm \in STM$.

Let *UIU* be the set of user interface units in the software.

Let *ADU* be the set of adapter units in the software.

Let *FCU* be the set of façade units in the software.

This principle is violated if

$$\exists\, stm\ ((|FSMTU_{stm} - F| + |IFSMTU_{stm} - IF| + |MSMTU_{stm} - M| + |IMSMTU_{stm} - IM|) > (|FSMTU_{stm} \cap F| + |IFSMTU_{stm} \cap IF| + |MSMTU_{stm} \cap M| + |IMSMTU_{stm} \cap IM|)) \wedge (U \notin (UIU \cup ADU \cup FCU))$$

Measurement example:

Class C2

$F = \{s\}$

$IF = \{\}$

$M = \{gets(), main()\}$

$IM = \{\}$

$STM = \{main()\}$

$FSMTU_{main()} = \{\}$

$IFSMTU_{main()} = \{\}$

$MSMTU_{main()} = \{greeting()\}$

$IMSMTU_{main()} = \{\}$

$UIU = \{\}$

$ADU = \{\}$

$FCU = \{\}$

$((|FSMTU_{main()} - F| + |IFSMTU_{main()} - IF| + |MSMTU_{main()} - M| + |IMSMTU_{main()} - IM|) > (|FSMTU_{main()} \cap F| + |IFSMTU_{main()} \cap IF| + |MSMTU_{main()} \cap M| + |IMSMTU_{main()} \cap IM|)) \wedge (C2 \notin (UIU \cup ADU \cup FCU))$ // Detected: class C2 violates Feature Envy.

Freeloader (a.k.a. Lazy Class): A class that isn't doing enough to pay for itself should be eliminated.

Source code:

```
public class C2 {

}
```

Violation check expressions:

Consider a unit U.

Let *F* be the set of fields of U.

Let *IF* be the set of fields of U.

Let *M* be the set of methods of U.

Let *IM* be the set of inter-type methods of U.

Let *P* be the set of pointcuts of U.

Let *AV* be the set of advices of U.

This principle is violated if

$$F \cup IF \cup M \cup IM \cup P \cup AV = \emptyset$$

Measurement example:

Class C2

$F = \{\}$

$IF = \{\}$

$M = \{\}$

$IM = \{\}$

$P = \{\}$

$AV = \{\}$

$F \cup IF \cup M \cup IM \cup P \cup AV = \{\}$ // Detected: class C2 violates Freeloader.

Speculative Generality: Often methods or classes are designed to do things that in fact are not required. The dead-wood should probably be removed.

Source code:

```
public class C1 {
        public void greeting(String name){
                System.out.println("Hello Ms. " + name);
        }
        public void testGreeting(){
                greeting("Test");
        }
}
public class Main {

        public static void main(String[] args) {
                C1 c1 = new C1();
                c1.greeting("Sunnie");
        }

}
```

Violation check expressions:

Consider a unit U.

Let *FT* be the set of members (methods/ inter-type methods/ constructors) of U.

Let UCM_{ft} be the set of units calling to a function *ft* where $ft \in FT$.

Let *P* be the set of pointcuts of U.

Let J_p be the set of corresponding join points of a pointcut *p* where $p \in P$.

This principle is violated if

$$(FT \neq \emptyset) \wedge \exists ft\ (UCM_{ft} = \emptyset) \wedge \exists p\ (J_p = \emptyset)$$

Measurement example:

Class C1

FT = { greeting(String name), testGreeting()}

$UCM_{greeting(String\ name)}$ = {Main}

$UCM_{testGreeting()}$ = { }

$(FT = \{ greeting(String\ name), testGreeting()\}) \wedge (UCM_{testGreeting()} = \emptyset)$ //

Detected: class C1 violates Speculative Generality.

Message Chains: Occur when you see a long sequence of method calls or temporary variables to get some data. This chain makes the code dependent on the relationships between many potentially unrelated objects.

Source code:

```
public class Main {

        public static void main(String[] args) {
                C1 c1 = new C1();
                System.out.println(c1.getA().getB().getC().getD().getE().doIt());
        }
}
```

Violation check expressions:

Consider a unit U.

Let *FT* be the set of functions (methods/ inter-type methods/ advices) of U.

Let ST_{ft} be the set of statements of a function *ft* where $ft \in FT$.

Let MSG_{stft} be the set of messages contained in a statement *st* of a function *ft* where $st \in ST$.

This principle is violated if

$$\exists st \exists ft(|MSG_{stft}| > 6)$$

Measurement example:

Class Main

FT = {main()}

$ST_{main()}$ = { C1 c1 = new C1();,

System.out.println(c1.getA().getB().getC().getD().getE().doIt());}

$MSG_{C1\ c1\ =\ new\ C1();}$ = { new C1()}

$MSG_{System.out.println(c1.getA().getB().getC().getD().getE().doIt());}$ = {doIt(), getE(), getD(), getC(), getB(), getA(), System.out.print()}

$|MSG_{System.out.println(c1.getA().getB().getC().getD().getE().doIt());}| = 7$ // Detected: class Main violates Message Chains.

Large Aspects: Whenever an aspect tries to deal with more than one concern, it could be divided in as many aspects as there are concerns.

Source code:

```
public aspect A1 {
        private String C1.name = "C1";
        private int C2.number = 2;

        pointcut callDo2(C1 c1): call (void source44.C1.do2()) && target(c1);
        before(C1 c1): callDo2(c1){
                do2a();
                System.out.println(c1.name);
        }

        pointcut callDo3(C2 c2): call (void source44.C2.do3()) && target(c2);
        before(C2 c2): callDo3(c2){
                do3a();
                System.out.println(c2.number);
        }

        public void do2a(){
                System.out.println("Do2a");
        }
        public void do3a(){
                System.out.println("Do3a");
        }

}
```

Violation check expressions:

Consider a unit U.

Let *NSF* be the set of non-static fields of U.

Let *NSIF* be the set of non-static inter-type fields of U.

Let *FT* be the set of functions (methods/ inter-type methods/ advices/ constructors) of U.

Let $FTUFIF_{nsfx}$ be the set of functions using nsf_x where $nsf_x \in (NSF \cup NSIF)$.

Let $FTUFIF_{nsfy}$ be the set of functions using nsf_y where $nsf_y \in (NSF \cup NSIF)$-$\{nsf_x\}$).

Let $FTUFT_{ftufifnsfx}$ be the set of functions using *ftufifnsfx* where *ftufifnsfx* $\in$ $FTUFIF_{nsfx}$.

Let $FTUFT_{ftufifnsfy}$ be the set of functions using *ftufifnsfy* where *ftufifnsfy* $\in$ $FTUFIF_{nsfy}$.

This principle is violated if

$$((|NSF \cup NSIF| > 1) \wedge (|FT| > 1)) \wedge (\exists nsfx \exists nsfy\ ((FTUFIF_{nsfx} \neq \emptyset) \wedge (FTUFIF_{nsfy} \neq \emptyset) \wedge (FTUFIF_{nsfx} \cap FTUFIF_{nsfy} = \emptyset)) \wedge \forall ftufifnsfx\ \forall ftufifnsfy\ ((ftufifnsfy \notin FTUFT_{ftufifnsfx}) \wedge (ftufifnsfx \notin FTUFT_{ftufiffnsfy}) \wedge (FTUFT_{ftufifnsfx} \cap FTUFT_{ftufiffnsfy} = \emptyset)))$$

Measurement example:

Aspect A1

$NSF = \{\}$

$NSIF$ = {C1.name, C2.number}

FT = { before(C1 c1): callDo2(c1), before(C2 c2): callDo3(c2), do2a(), do3a()}

$FTUFIF_{C1.name}$= { before(C1 c1): callDo2(c1) }

$FTUFIF_{C2.number}$= { before(C2 c2): callDo3(c2) }

$FTUFT_{ftufifC1.name} = \{\}$

$FTUFT_{ftufifC2.number} = \{\}$

$((|NSF \cup NSIF| = 2) \wedge (|FT| = 4)) \wedge ($ ($FTUFIF_{C1.name}$= { before(C1 c1): callDo2(c1) }) $\wedge$ ($FTUFIF_{C2.number}$= { before(C2 c2): callDo3(c2) }) $\wedge$ ($FTUFIF_{C1.name} \cap FTUFIF_{C2.number} = \emptyset$) $\wedge$ (before(C2 c2): callDo3(c2) $\notin$ $FTUFT_{ftufifC1.name}$) $\wedge$ (before(C1 c1): callDo2(c1) $\notin$ $FTUFIF_{C2.number}$) $\wedge$ ($FTUFT_{ftufifC1.name} \cap FTUFIF_{C2.number} = \emptyset$))) // Detected: aspect A1 violates Large Aspects.