Knowledge driven verification of positional tolerances

G. Concheri [(a)], *V. Milanese* [(b)]

[(a)] Università di Padova – Dip. ICEA, Lab. di Disegno e Metodi dell'Ingegneria Industriale
[(b)] Università di Udine – DiMI – Dipartimento di Matematica e Informatica

Article Information

Keywords:
Knowledge Based Engineering
Geometrical Tolerances
Paper gaging
Design Patterns
Java

Corresponding author:
Gianmaria Concheri
Tel.: +39 049 8276739
Fax.: +39 049 8276738
e-mail: gianmaria.concheri@unipd.it
Address: via Venezia, 1 – 35131 Padova (Italy)

Abstract

Geometrical specification and verification of products is achieving a more and more important role in Industry .
In case of specifications associating modifiers to positional tolerances and datums, the direct conformance verification of a workpiece is not immediate. Besides advanced functions of CMM software, which usually only indicate conformance or non-conformance to specifications, a more detailed analysis of a specific workpiece can be performed using manual or graphical computations as e..g. in "paper gaging" techniques.
Aim of this work is to present a prototypal knowledge based module implementing the tolerance analysis procedure and the conformance verification rules.
This module, implemented in Java, automatically generates a graphical representation of the tolerance zones on which a rule based tolerance analyzer infers the conformance/non-conformance conditions, allowing the operator to identify and possibly improve the fabrication process. The software module uses a design patterns approach to structure the overall architecture in order to simplify knowledge reuse and promote modularization.

1 Introduction

The ISO Geometrical Product Specification (GPS) standards [1], and the related ASME Geometric Dimensioning and Tolerancing (GD&T) standard [2], are very powerful tools intended for rigorous, concise and unambiguous specification of geometry, permissible deviations and control procedures of industrial products. Using the GPS or GD&T languages it is possible to clearly state the functional requirements and the mutual relationships among the geometric features of a product in the related Technical Product Documentation (TPD).

Among others, position tolerance specification of holes and shafts, especially in conjunction with the maximum material requirement (MMR) [3], e.g. in 2D layouts of flanged couplings, pipe flanges, etc., results in a very effective indication of coupling requirements. It allows for the reduction of scraps while preserving product functionality and quality, when compared with the traditional, non geometrical, tolerancing approach. Despite the benefits, geometrical position tolerancing with MMR applied to toleranced features or datums which are Feature of Size (FoS), is not very common in TPD. Besides the cultural/educational barriers that many mechanical design professionals may rise against the MMR use, the actual verification of MMR specifications may be not so straightforward. MMR was originally proposed in the fifties in USA to accurately state the actual behaviour of functional gages. Functional gages indeed constitute the embodiment of the virtual condition boundary, according to the design specifications, and therefore allow a direct control of the part. The manufacture of a functional gage is usually expensive and each gage is specifically designed for testing a distinct part. The control technique based on functional gage is very fast, but not flexible at all, because even small changes in tolerance specifications usually force to the complete re-manufacture of the gage. They are well suited for on line check of mass produced parts.

Currently, the instruments mainly used for accurate quality control of industrial products are Coordinate Measuring Machines (CMMs) that investigate the part geometry by sampling discrete sets of points on the different features in order to extract the significant geometrical parameters. CMMs are very expensive, time consuming and require skilled and experienced operators, but they are very flexible measurement instruments, and the attainable results may be very precise. CMMs are usually well suited to accurate investigation of single objects or parts produced in small series. In CMMs, a dedicated analysis software is usually available that processes the raw set of sampled points according to proprietary procedures in order to decide part conformance to specifications.

In investigating the data collected by means of CMMs, in order to assess part compliance with the stated tolerances in a clear and disclosed manner, the use of graphical techniques, usually recognized as "paper gaging", is very appealing, also because of their easiness of use, the explicit visualization of deviations and therefore the suggestions on the extent of the hypothetical re-work as well as instructions for adjusting the manufacturing process. The direct application of the paper gaging procedure may help the operator to understand and assess the CMM generated data with better confidence than using proprietary analysis software. Paper gaging based inspection can take the place of component check by functional gages, with consequent elimination of the manufacture costs of the expensive physical gages.

The "graphical inspection techniques" or "paper gaging" subject is well covered in literature (see e.g. [4], [5], [6], [7], [8], [9], [10], [11]), and a detailed presentation can be found in [12]. According to [10], paper gaging is extremely useful in capturing "dynamic tolerances" related to datum

features subject to size variation or feature-to-feature relationships within a pattern of holes. The management of these cases is very complex, also using dedicated inspection software, so the ability to graphically manipulate the grid overlay allowed by the paper gage technique gives the inspector a way to simulate the actual tolerance effects. Providing a visual record of the actual produced features, paper gaging can be an extremely effective tool for evaluating process trends and identifying underlying problems. Unlike hard gages or typical inspection analysis software, which simply verify GO/NO-GO attributes of the workpiece, the paper gage can provide the operator with a clear illustration of production problems and the precise adjustment necessary to bring the process back into control. In addition, paper gaging techniques can be used also in the design phase to simulate the effects of the assumed tolerancing scheme.

According to literature (i.e. [10]), the primary drawback to paper gage method of verification is that it is much more labor-intensive than use of a physical functional gage. Assuming that a set of coordinates can be obtained by a CMM, the graphical representation requires extra time and the accuracy of the method is limited when performed manually. In addition, the interpretation of the graph requires some skills and experience by the operator. The above considerations suggest the need of a dedicated software tool for directly managing the CMM generated data in order to automatically produce the graph and to support the operator in inferring the correct interpretation of data. Aim of this work is to present a prototypal knowledge based module implementing the tolerance analysis procedure detailed in [12] and a preliminary set of conformance verification rules. This module, implemented in Java, uses a “design patterns” approach [13] to structure the overall architecture in order to simplify knowledge reuse and promote modularization.

2 Software implemetation

2.1 Design patterns

At present, design patterns provide an assessed methodology for software projects development which is capable of formalizing, in an abstract and general way, programming aspects arising and being significant in different application environments. As such, they provide realization schemata which may tailored to specific applications.

Their abstract definition avoids strict implementation constraints and, therefore, it may be easily mapped to different programming paradigms, still imprinted to the object oriented approach. This approach clearly defines the separation between classes, as defined in C++, Prolog++ or Java, where they are considered to be models for data structures, and the rule-based inference of properties stated by languages such as Lisp or Prolog.

A field where design patterns may be profitable used is the development of software tools devoted to control the construction of mechanical components. In the problem considered in this paper, concerning with verification of positional tolerancing involved by mechanical work-cycles, what outlined in the previous section suggests using an *Observer pattern*: it relies on observed entities which, at run time, are selectively bound to one or more observers whose duty is collecting measures and synthesizing data coming from the former observed entities. The overall structure of the pattern is reported in figure 1, where it is described using UML (Unified Modeling Language) [14], a standard in the field of general-purpose modeling languages.

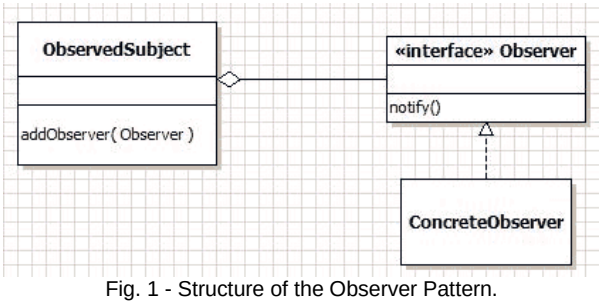

Fig. 1 - Structure of the Observer Pattern.

2.2 Software architecture

The software architecture of the pattern relies on a chain of classes, named *WorkDataNP*, *WorkDataTD* and *WorkDataAP*, which are separately devoted to model: the set of nominal positions of the mechanical components considered (*WorkDataNP*), the tolerance values admitted for such data (*WorkDataTD*), the actual values of the manufactured mechanical components (*WorkDataAP*). On the whole, the three classes provide an inheritance chain of properties which must characterize the objects (the mechanical components) that may be instantiated from class *WorkDataAP*. Instances of this class behave like *observed* objects, in what regards values of the features they own, which may be considered and analyzed by one or more *observers*. These last are provided by suitable classes having the duty to integrate information coming from the class *WorkDataNP* and its subclasses with further data synthesized from other available information sources according to the work and design specimens used.

Every object instantiated from *WorkDataAP*, therefore, on one side requires to be considered as a simple container of information concerning features related to the work process, on the other to be handled as a source of data, nominal (acquired by inheritance) and measured, to be subsequently integrated with more data pertaining to other computational units.

The former constraint forces the class *WorkDataNP*, root of the modeling hierarchy, to encapsulate:

- a suitable data structure capable of supporting the nominal, tolerancing and constructive information which characterize each feature; such information must be separately acquired by means of specific methods pertaining to the different (sub)classes of the hierarchy, in this way providing a polymorphic group of data acquisition functions;
  the latter aspect, related to exportability of information towards observer classes which may only be known at run time, obliges the class *WorkDataNP* itself and its subclasses to guarantee their instances
- the property to be handled as objects observable in what regards their state changes.

From an operational viewpoint the last stated property requires the definition of a suitable interface, named *Observable*, to force, within classes of type *WorkData*, the implementation of methods which are mandatory to ensure the functional specimen of exporting data, in an exclusive and selective manner, towards one or more observation classes.

On their whole, the classes of group *WorkData* and interface *Observable* may be modelled by the UML diagram outlined in figure 2. It is noticeable that observed objects may be instantiated only from the class *WorkDataAP* but their features may be modified at any level of the class hierarchy.

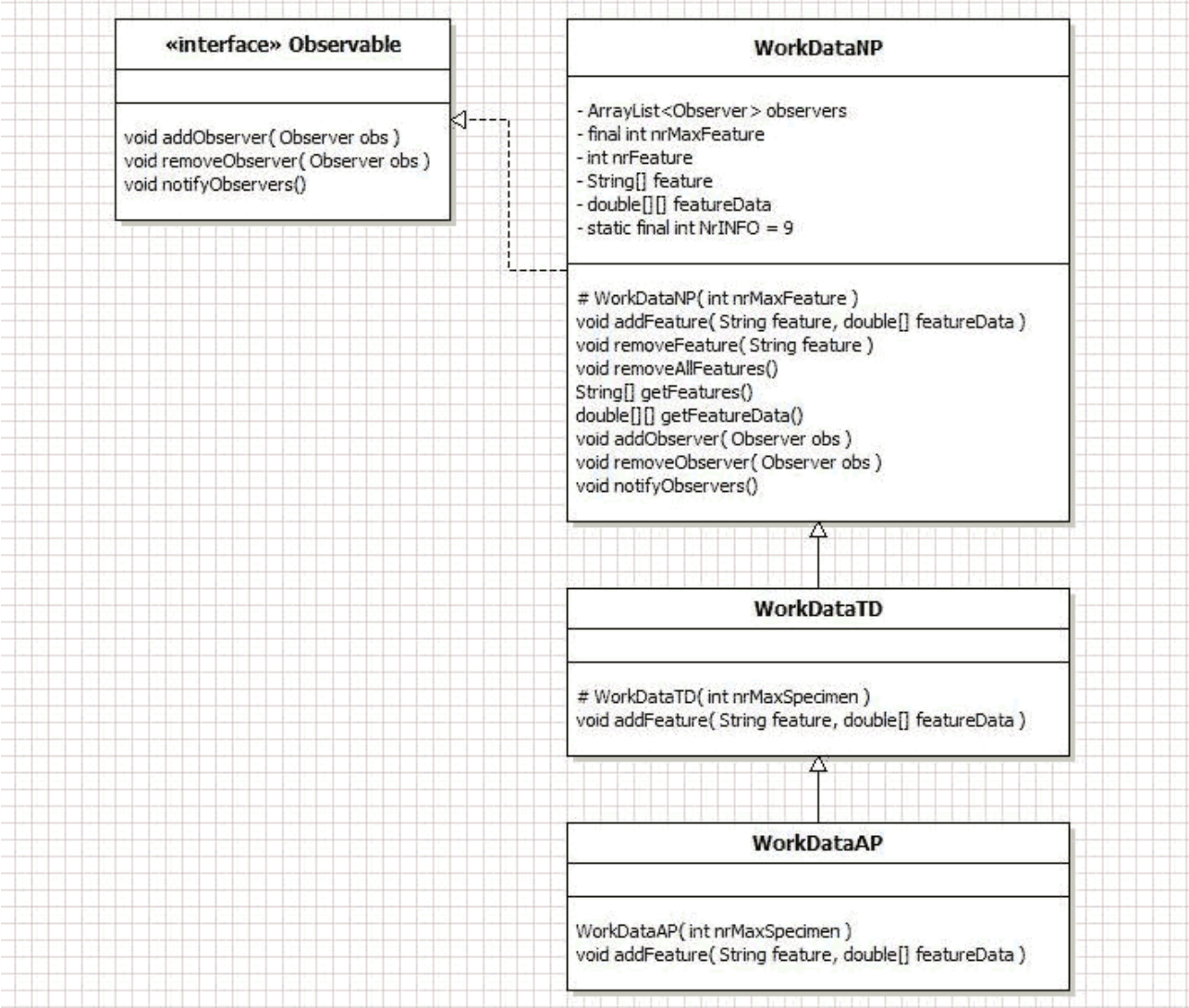

Fig. 2 – UML diagram of observable classes.

The Java pseudocode depicting the functionality and data structures of classes *WorkData* is given in Appendix.

Observation classes, the observers will have origin from, work in a symmetrical way: they are devoted to realize diversified policies in order to state the tolerancing regions which will be inferred from data made available by the observed objects or which are related to further possible specimens required by the different methodologies used to design the mechanical components or to evaluate the output data. The typical architecture of observer classes is sketched in figure 3.

From the availability of different and possibly coexistent computational models the requirement rises of modelling the whole set of observation classes by means of an inheritance tree which is rooted in a common superclass, named *ObserverUnit*, characterized by the following properties:

- to be used as a mandatory observer of observable entities, in other words to provide an implementation for the following

```
interface Observer {
    public void update( Observable observable );
}
```

- to encapsulate the operations which are common to all models of observers, postponing to its subclasses, in a fixed route, the realization of any particular and personalized implementation of methods pertaining to the observers.

As a consequence, the class *ObserverUnit* must be defined as an **abstract** class, in order not to be directly instanceable, and it must have a method *update(Observable)* at its disposal whose code is only implemented in what concers the acquisition of data pertaining to the features to be handled and that are provided by the observed instances of the class *WorkDataAP*; any further specialized code to be activated is competence of the **abstract** method *selectiveUpdate()* which will be reconfigured in the specialized observers' subclasses.

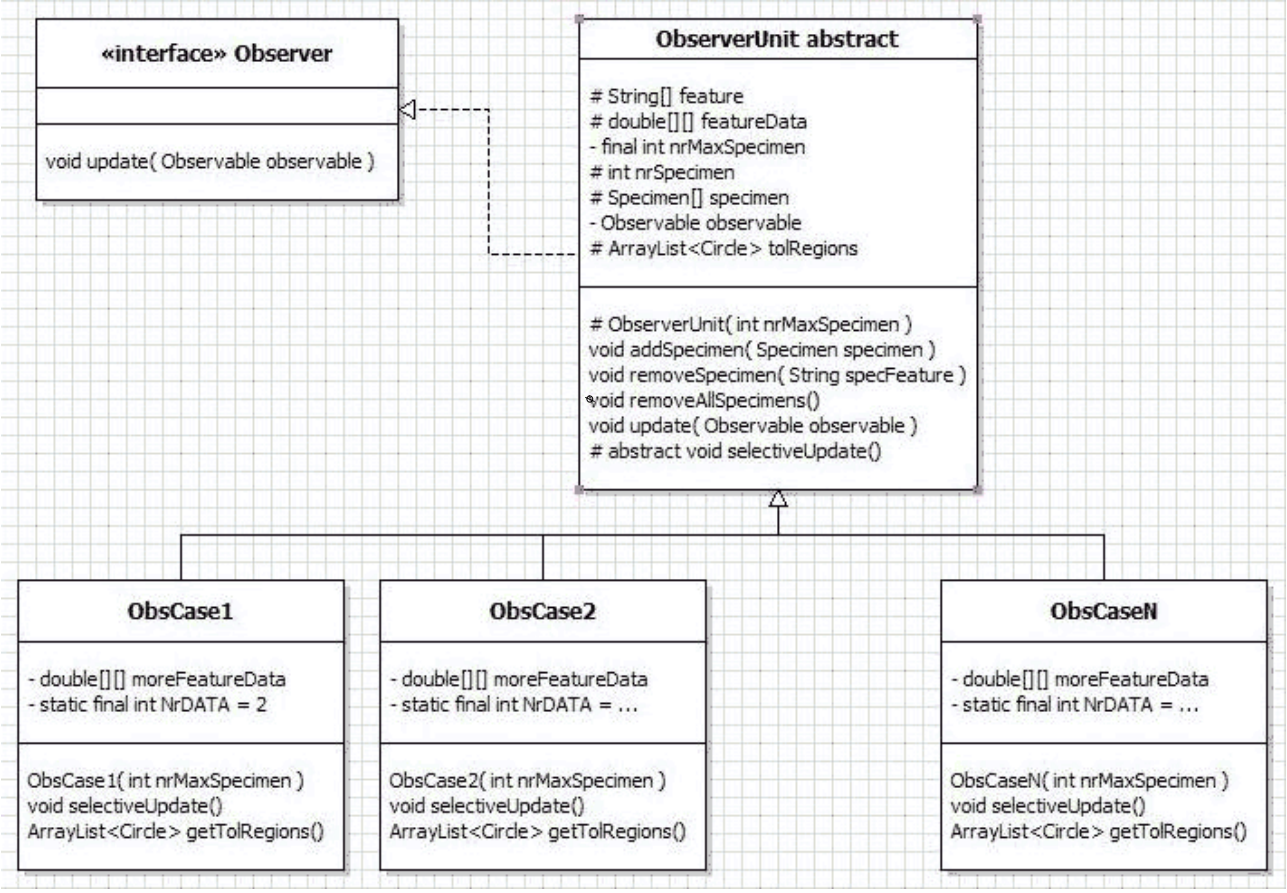

Fig. 3 – UML diagram of observer classes.

With regard to the last consideration outlined, it must be noted that information characterizing the semantics of any particular observer are acquired by means of method *addSpecimen(Specimen)* and recorded in the array *specimen[]*. See Appendix for datails on class *Specimen* and the types of data it uses.

The further container *tolRegions*, of type *ArrayList<Circle>*, is then devoted to support the tolerance regions which will be recognized by the specialization subclasses of *ObserverUnit* to which the realization of the specific implementation policies is postponed.

The objects it encloses (tolerance regions) are to be used by the application environment to evaluate the acceptability of data pertaining to the mechanical components built. Such an environment could then provide a bridge between the work data, considered at the light of the tolerancing model in use, and a graphical interface where the tolerance regions would be drawn giving the user a useful visual check of data acceptability. Also in this eventuality, the whole software architecture could follow a well accepted scheme Model-View-Controller, where the control could be based on a suite of menus put at user's disposal.

3 Examples of graphical analysis

In order to verify the functionality of the Java module, the test-case depicted in fig. 4 has been analyzed. In this figure a round part is presented together with three different sets of tolerance specifications for datum B and the four holes respectively.

Assuming that a single manufactured component is selected and measured in order to verify conformance or not-conformance to tolerance prescriptions, both the nominal and the measured data for datum B and the four holes are reported in table 1.

3.1 Case 1 - Tolerance specifications without material condition modifiers

The conformity of the considered part can be checked by plotting to scale on a graph, for each hole, the relevant circular tolerance zone (Ø 0.25 in case 1) centered on the actual deviation from the exact position (εx and εy values in table 2), and verifying if all the circular tolerance zones contain the plot origin (fig. 5). The assessment of the compliance with the stated tolerances requires:

- the computation of the possible common intersection area among all the different tolerance zones;
- the check that the common intersection area contains the plot origin.

In the example presented it is immediate to recognize that neither a common intersection area exists among the various tolerance zones, nor the tolerance zones contain the plot origin, and therefore the stated tolerances are not satisfied.

3.2 Case 2 - Tolerance specifications with MMR specified for the four holes only

If the maximum material requirement is applied to hole position tolerance, the position tolerance zone for each hole is increased by the “bonus” defined, for holes, as the difference between actual size and maximum material size. In the graph in fig. 6 the circumferences representing the allowed tolerances are drawn each centered at the

corresponding actual hole center location. In this case, a common intersection area exists among the four circumferences, but the plot origin is not contained in it, therefore stating non conformance.

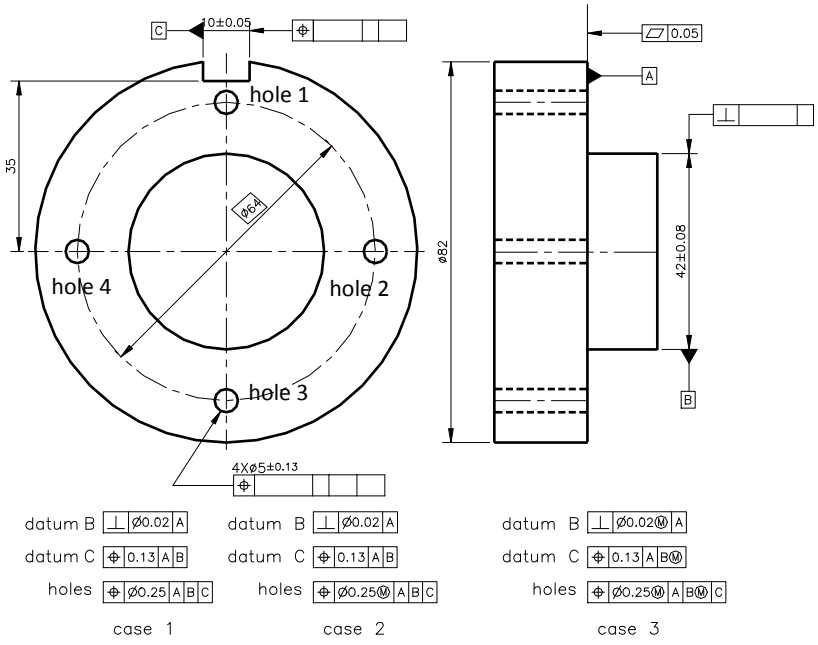

Fig. 4 – Flanged component with different tolerance specification sets

Feature	nominal position [mm]			spec. toler. deviation [mm]			actual position [mm]		
	x	y	size	lower	upper	position	x	y	size
datum B	0.0	0.0	Ø 42.	-0.08	0.08	0.02*	0.00	0.00	Ø 41.99
hole 1	0.0	32.0	Ø 5.0	-0.13	0.13	0.25	-0.18	32.16	Ø 5.13
hole 2	32.0	0.0	Ø 5.0	-0.13	0.13	0.25	32.11	0.04	Ø 4.97
hole 3	0.0	-32.0	Ø 5.0	-0.13	0.13	0.25	0.13	-31.93	Ø 5.03
hole 4	-32.0	0.0	Ø 5.0	-0.13	0.13	0.25	-31.91	-0.15	Ø 4.95

* for datum B, a perpendicularity tolerance has been specified instead of position tolerance.

Tab. 1 - Nominal and measured position and size for toleranced features in fig. 4

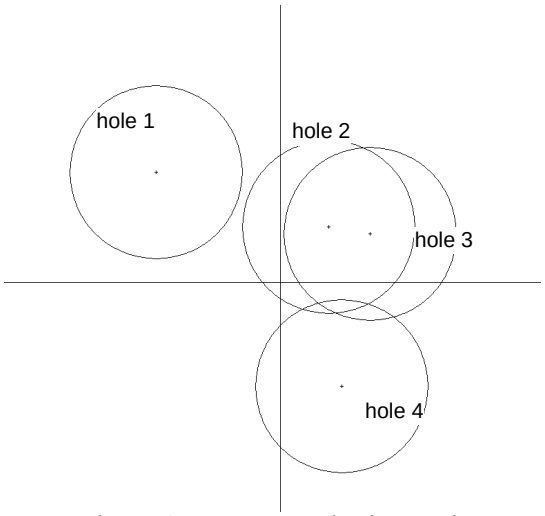

Fig. 5 – Case 1 paper gaging (no MMR)

Feature	εx	εy	bonus tolerance (with MMC)	allowed tolerance (with bonus)
datum B	0.00	0.00	0.09	0.11
hole 1	-0.18	0.16	0.26	0.51
hole 2	0.07	0.08	0.03	0.28
hole 3	0.13	0.07	0.16	0.41
hole 4	0.09	-0.15	0.08	0.33

All dimensions in mm.

Tab. 2 – Computed tolerance parameters

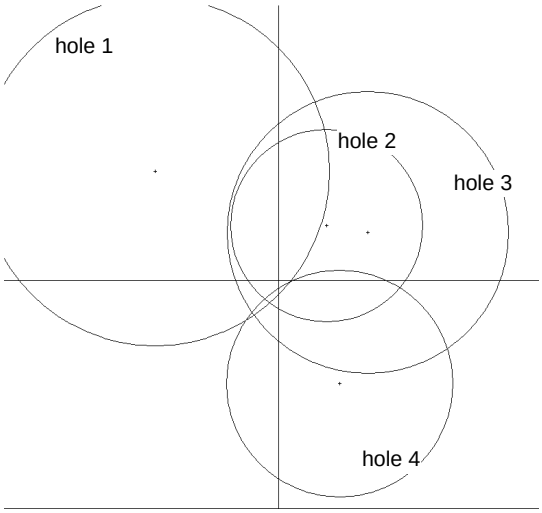

Fig. 6 – Case 2 paper gaging (MMR only on holes)

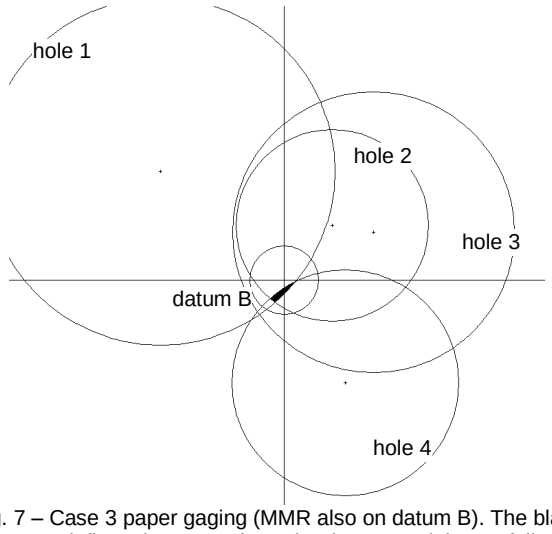

Fig. 7 – Case 3 paper gaging (MMR also on datum B). The black area defines the zone where the datum B origin can fall.

3.3 Case 3 - Tolerance specifications with MMC specified for datum B and the four holes

If the MMR is applied to the datum B, contrary to the previous case, the produced component is no longer intended to be fixed by clamping the pin (datum B), but allowing it to float inside its virtual condition boundary, whose diameter can be easily computed as:

virtual size = MM size + perpendicularity tolerance
= 42.08 + 0.02 = 42.10 mm (1)

Assuming a perpendicuarity deviation of 0.01 mm, the actual virtual size for datum B is easily computed as:

actual virtual size =
= actual size + perpendicularity deviation = hole 4
= 41.99 + 0.01 = 42.0

Therefore, the gap between virtual condition and actual virtual size for datum B, called “shift allowance”, is:

shift allowance
= virtual condition - actual virtual size
= 42.1 - 42.0 = 0.1 mm

This gap defines the allowable “shift” for datum B and can be represented on paper gage graph as the circular area inside which the plot origin can be moved in order to accommodate the four hole positions. In figure 7, computed paper gage is presented for case 3. The intersection among holes tolerance zones falls inside the datum B shift allowance circle, therefore the component specified according case 3 is acceptable.

4 Conclusions

The prototypal knowledge based module has proved to be an interesting test-bed for developing more complex and articulated implementation of rule driven tools. The design pattern approach could be further investigated in order to realize environments which are suited to support interchangeable modules oriented to face the control of tolerancing problems.

Appendix

This appendix reports in a concise manner the most significant classes used by the software and their possible mutual dependencies as introduced in section 2. Classes are detailed described in relation to the data the instantiated object enclosed and to the required protection level (private or protected). On the contrary, methods are simply provided with a synthetic description of their pseudo-code, in order to emphasize their principal functionalities and goals.

```
interface Observable {

    public void addObserver (Observer obs);                    (2)

    public void removeObserver (Observer obs);

    public void notifyObservers();
}                                                              (3)

class WorkDataNP implements Observable {

    private ArrayList<Observer> observers;
    private final int nrMaxFeature;
    private int nrFeature;
    private String[] feature;
    private double[][] featureData;
    private static final int NrINFO = 9;

    protected WorkDataNP( int nrMaxFeature ) {
        // Creates arrays feature[] and
featureData[][] containing
        // the identifies of features and their nominal,
tolerance and real data
        // Creates an empty list of observers
    }

    public void addFeature( String feature, double[]
featureData ) {
        // Adds or modifies data of a specific feature in
feature[] and featureData[][]
    }

    public void removeFeature( String feature ) {
        // Removes the specific feature from feature[]
and featureData[][] if present
    }
```

```
    public void removeAllFeatures() {
        // Removes all features from feature[] and
featureData[][]
    }

    public String[] getFeatures() {
        // Exports all identifies existing in feature[]
    }

    public double[][] getFeatureData() {
        // Exports all featureData[][] values related
to features
    }

    public void addObserver( Observer obs ) {
        // Adds a new observer to observers
    }

    public void removeObserver (Observer obs) {
        // Removes the specified observer from
observers
    }

    public void notifyObservers() { }
        // Notifies observers the feature[] and
featureData[][] values when required
    }
}


class WorkDataTD extends WorkDataNP {

    protected WorkDataTD( int nrMaxFeature ) {
super( nrMaxFeature ); }

    public void addFeature( String feature, double[]
featureData ) {
        // Adds or modifies tolerance data of a specific
feature in featureData[][]
    }
}


class WorkDataAP extends WorkDataTD {

    public WorkDataAP( int nrMaxFeature ) {
super( nrMaxFeature ); }

    public void addFeature( String feature, double[]
featureData ) {
        // Adds or modifies actual data of a specific
feature in featureData[][]
    }
}


interface Observer {

    public void update( Observable observable );
}


abstract class ObserverUnit implements Observer {

    protected String[] feature;
    protected double[][] featureData;
    private final int nrMaxSpecimen;
    protected int nrSpecimen;
    protected Specimen[] specimen;
    private Observable observable;
    protected ArrayList<Circle> tolRegions;

    protected ObserverUnit( int nrMaxSpecimen ) {
        // Creates array specimen[] containing the
specimens of features used
    }

    public void addSpecimen( Specimen specimen ) {
        // Adds or modifies data of a specific specimen
in specimen[]
    }

    public void removeSpecimen( String specFeature
) {
        // Removes the specific specimen type from
specimen[]
    }

    public void removeAllSpecimens() {
        // Removes all specimens from specimen[]
    }

    public void update( Observable observable ) {
        // Records the observed object, source of data
for this observer
        // getting its features[] and featureData[][] to
subsequent use
        // Activates a more selective update according
to the observer used
        // delaying the implementation to a specific
subclass
    }

    protected abstract void selectiveUpdate();
}


class ObsCase1 extends ObserverUnit {

    private double[][] moreFeatureData;
    private static final int NrDATA = 2;

    public ObsCase1( int nrMaxSpecimen ) { super(
nrMaxSpecimen ); }

    public void selectiveUpdate() {
        // Allocates more space for computed data
required by the specific observer
        // Specific operations for the Observer of case
1
        // .. calculates data for moreFeatureData[][]
            ... ...
        // .. creates tolerance regions in tolRegions
            ... ...
    }

    public ArrayList<Circle> getTolRegions() {
        return tolRegions;
    }
}


class Specimen {

    private String feature;
    private String typeString;
    private DimField dimField;
    private int nrRef;
```

```
    private RefField[] refField;

    public Specimen( String feature, String typeString, DimField dimField, RefField[] refField ) {
        // Creates a specimen for the specified feature type
    }

    // Public methods devoted to exporting current values
    // of feature, typeString, dimField and related subfields, refField

    public String toString() { // Returns a string description of the specimen }
}


class RefField {

    protected String refChar;
    protected String mmc;

    public RefField( String refChar, String mmc ) {
        // Creates a mmc-specimen for the feature type
    }

    public void setMMC( boolean on ) { // Sets the current mmc value }

    public String toString() { // Returns a string description of the mmc value }
}


class DimField {

    protected String sym;
    protected double value;
    protected String mmc;

    public DimField( String sym, double value, String mmc ) {
        // Creates a dim-specimen for the feature type
    }

    public void setMMC( boolean on ) { // Sets the current mmc value }

    public String toString() { // Returns a string description of the dim value }
}
```

References

[1] ISO TC/213 standards http://www.iso.org/iso/home/store/catalogue_tc/catalogue_tc_browse.htm?commid=54924 accessed 30th Apr 2013.

[2] ASME Y14.5-2009 - Dimensioning and Tolerancing, ASME, 2009

[3] ISO 2692:2006 GPS - Geometrical tolerancing - Maximum material requirement (MMR), least material requirement (LMR) and reciprocity requirement (RPR), ISO standards, 2006

[4] Foster L.W., Geo-Metrics III, Addison-Wesley, 1994

[5] Meadows, J.D., Geometric Dimensioning and Tolerancing, Dekker, NY, 1995

[6] Neumann A., Geometric Dimensioning and Tolerancing, Technical Consultants Inc., 1995

[7] Wilson B.A., Design Dimensioning and Tolerancing, The Goodheart-Willcox Co., 1996

[8] Podda G., Le tolleranze secondo la normativa americana, in Chirone E., Tornincasa S., Disegno tecnico industriale, il capitello, Torino, 1997

[9] Meadows, J.D., Measurement of Geometric Tolerances in Manufacturing, Dekker, 1998

[10] Drake P.Jr., Dimensioning and Tolerancing Handbook, McGraw-Hill, 1999

[11] Henzold, G., Geometrical Dimensioning and Tolerancing for Design, Manufacturing and Inspection, 2nd Ed., Elsevier, 2006

[12] Baratto F., Concheri G., Position Tolerances Control Using Virtual Paper Gaging Techniques, XI ADM - Int. Conf. on Design Tools and Methods in Industrial Engineering. vol. D, p. 105-114, Palermo, December 8th-12th, 1999

[13] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – Design Patterns: Elements of Reusable Object-Oriented Software – Addison Wesley, 1995

[14] Craig Larman – Applying UML and Patterns – Prentice-Hall, 2001