Proceedings of ICAD2006
4th International Conference on Axiomatic Design
Firenze – June 13-16, 2006
ICAD-2006-30

AXIOMATIC DESIGN OF SHOP FLOOR PROGRAMMING SOFTWARE

Türker Oktay
toktay@adcole.com
Adcole Corporation
669 Forest Street
Marlborough, MA 01752 U.S.A.

ABSTRACT

Axiomatic Design methods were utilized in the architecture design of Shop Floor Programming Software for Computer Numerically Controlled (CNC) turning applications. Two very different design alternatives were evaluated based on Axiomatic Design. Although both alternatives complied well with the Object Oriented Programming (OOP) principles, the Axiomatic Design analysis helped choose the superior alternative. The resulting software architecture turned out to be more modular, easier to implement and easier to maintain.

Keywords: Axiomatic Design, Object Oriented Programming, Shop Floor Programming

1 INTRODUCTION

Computer Numerically Controlled (CNC) Machine tools are used for machining raw material stocks into finished shapes by executing a series of instructions called a part program. The part program commands a cutting tool through a pre-determined tool path to cut and shape the material. Although many excellent CAD/CAM software products are available in the market to automatically generate the part programs (usually referred to as G-Code), they tend to be too complicated and too time-consuming to use by small machine shops which tend to manufacture small lot sizes. Such machine shops prefer a sub-category of CAM software called "Shop Floor Programming" (SFP) software or sometimes called "Conversational Programming Software". The idea is literally to accomplish the CNC programming tasks in a shop floor environment, not in the engineering office. These software products accomplish much the same thing as full-fledged CAD/CAM products but they tend to be much easier-to-use and much faster to turn around fully-functional part programs [1-2]. Their lack of sophisticated features are well-compensated by their ability to turn out part programs quickly and efficiently. They offer more user-friendly user interfaces to operators, who have less training and less time to invest into learning a new CAD/CAM software package [3]. They are especially indispensable for use with CNC lathes and turning centers because turned parts are simpler than milled parts and they don't require the sophisticated features of full-fledged CAD/CAM systems.

The author of this paper and his associates designed and developed a Shop Floor Programming Software for CNC lathes. This paper outlines the author's experience and observations especially during the design phase. The SFP software developed here is add-on software to run on the CNC unit's CPU, not a separate stand-alone product. Since the CNC unit's main function is to execute parts programs in real time, it was important not to compromise the real time performance of the CNC by the demands of the shop floor program. Thus the SFP software was required to have a small footprint (use only a small amount of memory and hard disk space, be fast and efficient.) The author and his associates chose the Object Oriented Programming Technology (OOT) to make the software more efficient, more modular and easier to maintain. The Axiomatic Design principles were utilized to pick the best OOT framework for the software architecture.

2 BASIC ARCHITECTURE DESIGN

The primary objective of Shop Floor Programming (SFP) software is:

- to gather information from the user about the desired shape of the part, and the raw stock from which the part is to be made, and
- to generate a part program to convert a raw stock into a finished shape.

The part program generated by the SFP software is then executed by the CNC. The CNC machine tool moves the tools according to the instructions contained in the part program and removes material from the raw stock. When the execution of the part program is completed, the raw stock has been transformed into the finished part. This conversion process involves the application of several different types of machining (turning) processes. Figure 1 shows a typical turned part and some of the common turning operations such as Outside Diameter (OD) turning, Inside Diameter (ID) turning (Boring), Face turning, Drilling and Threading.

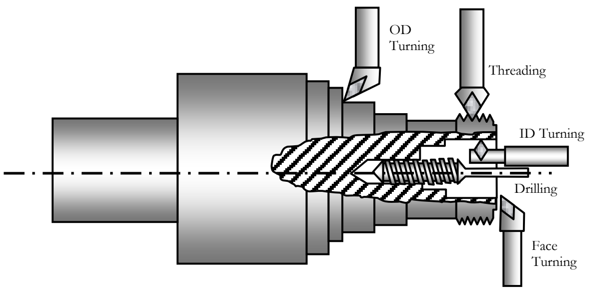


Figure 1 – A turned part and some of the common Turning Operations.

Each turning process involves multiple passes of the cutting tool on the raw stock. During each pass, only a small amount of material is removed. For the OD turning process, these passes are parallel to the cylindrical axis of symmetry. At each cutting pass, as material is removed, the cutting tool gets closer to the center of the part and the outer diameter is progressively reduced. For the ID turning process, the cutting passes are still parallel to the cylindrical axis of symmetry but cutting action starts from the part center and progresses outward at each pass. Facing process is perpendicular to the axis of symmetry. Drilling involves a drill bit which drills into the part and retracts periodically to allow the chips to fall, in order to prevent the chips from jamming the drill bit. Threading operation involves the formation of a helical thread on the inner or outer surface of the part. Threads are again formed by multiple successive passes in order to achieve the best thread quality. There are several other types of turning processes, not mentioned in this list. Most turning processes involve multiple passes of cuts where material is removed by a small amount each time. The amount of each cut (depth of cut) and the speed of the cutting tool (feed rate) are determined by the material properties of the raw stock. SFP software is expected to calculate the tool path for each successive pass in each turning process, effect the tool changes between the turning processes and convert the tool path into a part program which can be understood by the CNC. Figure 2 shows the functional requirements of SFP software in a data flow diagram format.

Examination of Figure 2 reveals two major functional requirements (FRs): One is to receive and display user input, and the other is to produce the tool path. Upon first glance, maintaining the independence of these two FRs seems relatively easy, provided an Object Oriented programming language such as C++ is used to implement the software.

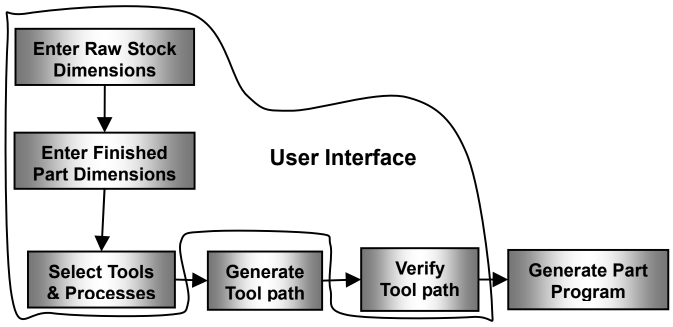


Figure 2 – SFP Software Data Flow Diagram

At the highest level of FR decomposition, the User Interface module (DP_1) and the Tool path Engine (DP_2) can be kept uncoupled from each other relatively easily.

$$\begin{Bmatrix} FR_1 \\ FR_2 \end{Bmatrix} = \begin{bmatrix} X & O \\ O & X \end{bmatrix} \cdot \begin{Bmatrix} DP_1 \\ DP_2 \end{Bmatrix}$$

Where

FR_1=Receive user input
FR_2=Generate tool path

And the DPs that satisfy the FRs are

DP_1=User interface
DP_2=Tool path engine

However, at this level of analysis of the design equation, it is impossible to identify the software objects and their methods, and translate those into actual implementation of the software code. Further decomposition of the FRs through a zigzagging process is necessary [4].

3 FUNCTIONAL REQUIREMENT (FR) DECOMPOSITION

FR_1, the capture of user input is accomplished via the User interface of the software program. The most important user interface element in modern operating systems (OS) is a window. Therefore, it is fairly typical in modern OOT-based software programs to have a software object called **Window** to represent the interactions between the user and the software program. This object often encapsulates such user-software interactions as prompting the user to enter data, accepting and organizing the data and after the data is processed by the software, displaying the results on the screen. Modern software development systems provide special software frameworks to implement the user interface in a standard and methodical way. Therefore the development of the user interface will be largely outside the scope of this paper. However, FR_2, the generation of tool path is specific to our case, thus it needs to be decomposed into smaller and more manageable requirements in order to develop the detailed design of the SFP software.

Object Oriented Programming encourages programmers to think of software modules as independent entities (objects) which contain data (attributes) and functions which operate on such data (methods). The more self-contained the objects are, the greater the amount of modularity that is achieved. Although it is not absolutely required, it is always advantageous to choose the software objects (classes) from among the objects we associate with the physical world. In order to identify the classes, Braude [5] recommends "listing every reasonable candidate class you can think of, and then aggressively pairing down the list to a few essential classes". By just looking at Figure 1, the turned **Part** emerges as the most reasonable and obvious software object for us to choose. Turning operations such as OD Turning, ID Turning, etc. can easily be imagined to represent the functions (methods) of this **Part** object. This choice definitely has the major advantage of maintaining a real life metaphor during the development the software, which is extremely helpful to the programmers during the implementation stage.

Figure 3 shows the data flow diagram between the two major objects in this software architecture, the **Window** object and the **Part** object. The **Window** object receives input from the user and converts it into a structured form and delivers it to the **Part** object. The **Part** object operates on that data by using its methods such as ODTurn(), IDTurn, Drill(), etc. As a result, the tool path is created; it is sent back to the Window object for a graphical display and is eventually sent to the CNC to manufacture new parts.

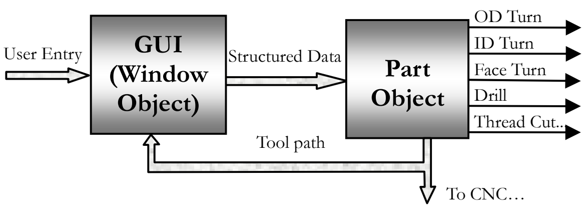


Figure 3 –Data flow diagram for the proposed software objects

Based on this architecture design, we can now decompose FR_2 (generate tool path) into more detailed FRs, each FR representing a separate turning process. The author and his team actually began writing the software code for the tool path engine, based on this software architecture but the presence of serious couplings between the FRs became quickly obvious and hampered the code development process. The axiomatic design equation for this software architecture can be shown below:

$$\begin{Bmatrix} FR_{21} \\ FR_{22} \\ FR_{23} \\ FR_{24} \\ FR_{25} \end{Bmatrix} = \begin{bmatrix} X & O & O & O & O \\ X & X & O & O & O \\ X & O & X & O & O \\ O & O & O & X & O \\ O & O & O & O & X \end{bmatrix} . \begin{Bmatrix} DP_{21} \\ DP_{22} \\ DP_{23} \\ DP_{24} \\ DP_{25} \end{Bmatrix}$$

FR_{21}=Generate OD cutting tool path
FR_{22}= Generate ID cutting tool path
FR_{23}= Generate face cutting tool path
FR_{24}= Generate drilling tool path
FR_{25}= Generate threading tool path

DPs are

DP_{21}= Part::ODTurn()
DP_{22}= Part::IDTurn()
DP_{23}= Part::FaceTurn()
DP_{24}= Part::Drill()
DP_{25}= Part::ThreadCut()

The Design Parameter (DP) vector here is populated by the functions (methods) of the **Part** object. *X*s show a coupling between the FRs and the corresponding DPs, while *O*s show instances where no coupling exists. Being members of the same **Part** object and being very similar turning operations, there is a great tendency for the ODTurn(), IDTurn(), and FaceTurn() methods to be coupled. These turning operations are fundamentally very similar. They all accomplish the removal of material from the part surface by successive parallel cuts (passes) using geometrically similar tools. It is natural that all these methods would share some common code. However, being function members of the same software object, there were no access restrictions to data among these functions, therefore the tendency for these functions to become intermingled (spaghetti-like code) was difficult to avoid.

Another concern about this architecture design was a second axiom concern. This design concentrated the entire tool path generation functionality of the software in the **Part** object. As a result, the information content was likely to be great, there would be very little periodicity [6] which would have a tendency to reduce the complexity of the design.

Both first axiom and second axiom concerns left us unsatisfied with this design and led us to seek a better design, one which was less coupled and simpler (lower information content).

4 AN UNCOUPLED DESIGN

When the reasons for coupling among the FRs of the original architecture were examined, it was found that the original choice for the software objects (DPs) did not lend itself to a sufficiently modular design and the amount of modularity that was achieved did not lead to functional independence [7]. In fact, all of the functionality of the Tool path Engine was concentrated within just one object, the **Part** object. Since OOT poses no access restrictions among the function members of the same object, there were many cross-references among the functions of the **Part** class, especially among the 3 similar turning processes.

Figure 4 shows the cutting patterns of the 3 similar turning processes. The Shop Floor Programming software needs to calculate each individual cutting pass based on the geometry of the raw stock and the finished part, and the material dependent cutting parameters such as the feed rate and the depth of cut. Notice that the OD turning, ID Turning and Facing operations are all similar to each other except for the direction of the cuts. All three turning processes make parallel cuts to remove material from the raw stock. This fact results in a sharing of the software code among the 3 turning algorithms. Sharing of the code alone is not necessarily an undesirable goal; however, the problem is, the functions which perform these 3 turning algorithms are not isolated from each other. As a result, interactions among the groups of code are not controlled. Even though OOT is being enforced [8], the choice for the software objects was made such that the isolation of the software objects was insufficient. This leads to coupling as well as increased complexity.

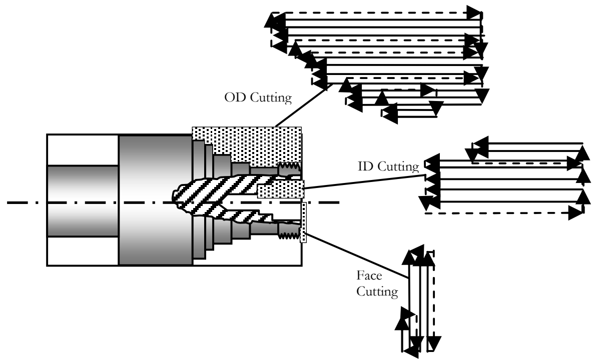


Figure 4 – Three different turning processes.

In order to achieve a better degree of software modularity, an uncoupled architecture design alternative was proposed. In this design, rather than defining an all-encompassing **Part** object which has function members that perform the various turning operations, separate individual **Process** objects were defined, each one representing another turning operation such as **ODTurn**, **IDTurn**, **FaceTurn**, **Drilling**, **Threading**, etc. processes. These objects are completely independent from each other and have no access to each other's data or methods. This ensures that no coupling is possible among the process-specific components of the Tool path Engine.

Figure 5 shows the data flow diagram for this proposed design. With this new architecture design, the turned part is no longer the main junction point of the software. When a user needs to generate a part program which only has an OD Turning operation involved, the only Process object that needs to be instantiated (created from a class blueprint) is the OD Turning object. This design helps keep the software code for each turning process separate, isolated and independent.

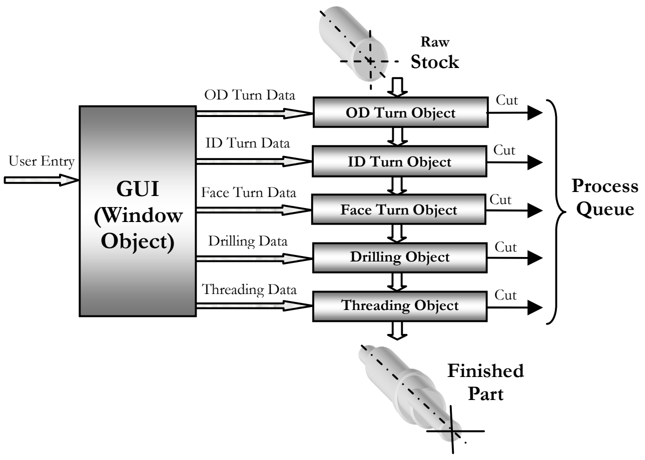


Figure 5 – Data Flow diagram among the objects in the uncoupled design.

As the desired part shape dictates it, more process objects are instantiated and added to a process queue. It is possible to have multiple process objects of the same kind occurring in multiple different places in the queue. The complete tool path can be generated simply by executing this queue in the specified order. The resulting design equation is fully uncoupled as shown below:

$$\begin{Bmatrix} FR_{21} \\ FR_{22} \\ FR_{23} \\ FR_{24} \\ FR_{25} \end{Bmatrix} = \begin{bmatrix} X & O & O & O & O \\ O & X & O & O & O \\ O & O & X & O & O \\ O & O & O & X & O \\ O & O & O & O & X \end{bmatrix} . \begin{Bmatrix} DP_{21} \\ DP_{22} \\ DP_{23} \\ DP_{24} \\ DP_{25} \end{Bmatrix}$$

The new DPs are

DP_{21}= ODTurn:Cut()
DP_{22}=IDTurn::Cut()
DP_{23}=FaceTurn::Cut()
DP_{24}=Drilling::Cut
DP_{25}=Threading::Cut()

The benefits of this architecture design are not solely limited to the First design axiom considerations, however. There is also an added benefit of reduced information content (Second axiom). Note that the tool path for each turning process is generated by a function simply called Cut(). Even though the software objects to which these functions belong are separate, they all have a function named with the same exact name and all the process objects are derived from the same parent object simply called **Process** via the inheritance mechanism of the C++ language (Figure 6). This is called polymorphism in the OOT parlance and it allows us to use the Cut() functions of any Process object in the same exact way without any regard to the particulars of that object. Basically each Process object is different in its internal details but it presents a common interface to the outside world. Taking advantage of OOT's polymorphism mechanism reduces complexity by introducing a form of periodicity to the software.

This periodicity not only ensures encapsulation but also facilitates the maintenance of the software and adding new features to the software. For example, if a new type of turning process is to be added, it can be derived from the same parent **Process** class as all the others and contain the same kind of Cut() function to generate the tool path as all the others. When adding a new process, there is no need to have a precise knowledge of any of the other turning processes. Axiomatic Design Theorem Soft 1 [6] states that "Uncoupled software or hardware systems can be operated without precise knowledge of the design elements (i.e., modules) if the design is truly an uncoupled design…" In the context of SFP software, this means a new turning process can be added without precise knowledge of the other turning processes.

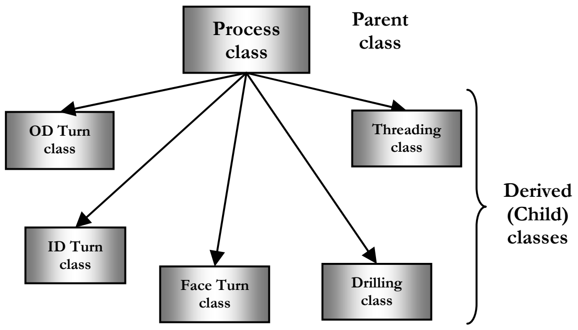
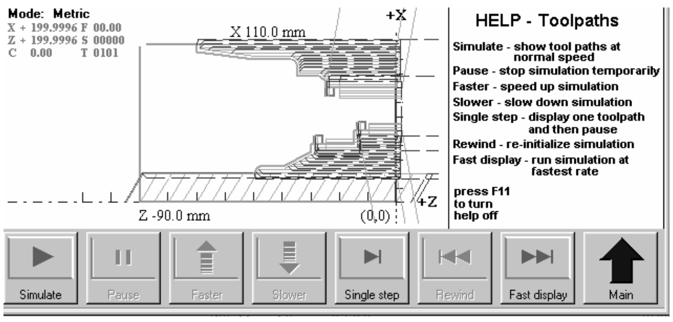


Figure 6 – Parent-child relationship among the classes.

Figure 7 shows a screen shot of the SFP software which was developed based on the chosen design. This screen displays the tool path generated by the software in 2-dimensions.



HELP - Toolpaths

Simulate - show tool paths at normal speed
Pause - stop simulation temporarily
Faster - speed up simulation
Slower - slow down simulation
Single step - display one toolpath and then pause
Rewind - re-initialize simulation
Fast display - run simulation at fastest rate

press F11 to turn help off

Figure 7 – Tool paths generated by the Shop Floor Programming software.

The software also has the capability to display a 3-D rendering of the part. As material is removed from the stock surface, the view of the part on the screen evolves step-by-step into the finished shape in a 3-dimensionally realistic form. (Figure 8)

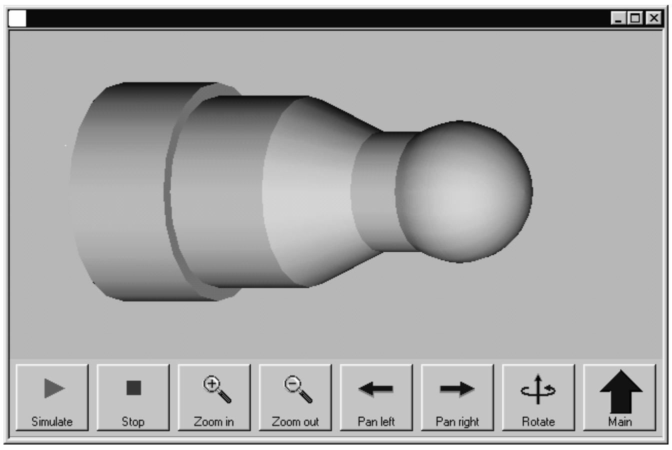


Figure 8 – 3-Dimensional simulation of the turning processes.

In addition to the 5 primary turning operations, several other turning processes were added to the SFP software over time: Grooving, Relieving, Copy turning, cut-off process, etc. Due to the modular design of the software, adding those processes was very easy and it did not increase the complexity of the software.

5 CONCLUSIONS

Object Oriented Software Technology (OOT) is a powerful method for developing a modular software design. This paper examined the author's experience with developing Shop Floor Programming Software by using OOT. It turned out that OOT alone did not ensure the selection of the best software architecture. There were at least two OOT alternatives which were equally valid in terms of their object-oriented structure but they were not equally desirable as design candidates. The advantage of the first proposed design was a much closer real life metaphor between the real-life objects and the software objects, namely the **Part** object. However, when this design was put to the test of the design axioms, it did not fare so well.

The second design was built around **Process** objects. Child objects such as **OD Turning**, **ID Turning**, etc. process objects were created via the inheritance mechanism of OOT. Each of these processes were different in their internal details but exhibited a uniform behavior to the outside world. This could be thought of as a form of periodicity which was lacking in the first design. As a result, the second design was superior both from the point of view of reduced coupling (1st axiom) and from the point of view of simplicity (2nd axiom).

The second design was also much more modular. It was quite easy to add a brand new turning process too the software because it could simply be added as yet another child process. Adding a new process didn't involve any interactions with the already-existing processes. As a result, maintaining the software and adding new processes were much easier in this design than the first proposed design.

6 REFERENCES

[1] Dickin, P. "Specialized CAM Takes Center Stage", *Desktop Engineering*, February 2006, Vol.11, No.6, pp. 24-28.

[2] Lewis, M. "Turning on CAM Technology", *American Machinist*, May 2002, Vol. 146, No.5, pp. 63-71.

[3] Evans, K., Polywka, J. and Gabrel, S. "Conversational Programming Considered", *Tooling & Production*, September 2001, pp. 70-73.

[4] Suh, N.P., Axiomatic Design: Advances and Applications, *Oxford University Press*, 2001.

[5] Braude, E.J. , Software Engineering: An Object-Oriented Perspective, *John Wiley & Sons, Inc.*, 2001, pp. 205-210.

[6] Suh, N.P., Complexity : Theory and Applications, *Oxford University Press*, 2005.

[7] Do, S.H. and Suh, N.P., "Object-Oriented Software Design with Axiomatic Design", *Proceedings of the 1st International Conference on Axiomatic Design*, Cambridge, MA, June 21-23, 2000.

[8] Clapis, P.J. and Hintersteiner, J.D., "Enhancing Object-Oriented Software Development Through Axiomatic Design" , *Proceedings of the 1st International Conference on Axiomatic Design*, Cambridge, MA, June 21-23, 2000.