OO Development and Maintenance Complexity

Daniel M. Berry
based on a paper by Eliezer Kantorowitz

Traditional Complexity Measures

Traditionally,

- **Time Complexity**
- **Space Complexity**
- **Both use theoretical measures, e.g.,** $O(N^2)$**, where** ***N*** **is some measure of input size.**

Algorithm Optimization

Algorithm optimization for time and/or space was a key concern back in good old days when the computer was the most expensive part of a computer installation, when machines were several orders of magnitude slower and smaller than the hand-held computer of today.

The traditional measures grew out of our attempt to understand algorithms and how to optimize them.

Nowadays

Nowadays, programmers are the most expensive part of a computer installation.

Programmmers spend a small part of their time developing new code and a large part of their time maintaining legacy code.

What We Should Measure

Therefore, we should measure the cost of development and maintenance complexity, and should be measuring program complexity relative to *programming* difficulty.

We would also like to use these complexities to predict the costs of program development and maintenance, the costs including, time to program, number of programmers needed, i.e., the time and space of development!

Extant Complexity Measures

There are a number of program complexity measures that purport to measure development and maintenance difficulty, including:

- **Lines of Code (LOCs)**
- **Cyclomatic (McCabe)**
- **Software Physics (Halstead)**

Extant Measures, Cont'd

See also, e.g.,

N. E. Fenton, *Software Metrics, a Rigorous Approach*, Chapman Hall, NY, 1991

B. Kitchenham, "Software Development Cost Models", *Software Reliability Handbook*, P. Rook (Ed), pp. 487–517, Elsevier, 1990.

Extant Measures, Cont’d

These measures are not very good.

There are always lots of anomalous exceptions, e.g., two programs of the same length, one with lots of nested loops and conditionals and one with no branches at all, are *not* of equal difficulty to program, understand, and maintain, yet they have the same number of LOCs.

Extant Measures, Cont’d

While experimental data have shown that there is a positive correlation between most of these measures and actual costs, the correlation is not good enough to use in prediction.

About all that can be said is that lengthening a program drives its costs up.

More Detailed Models

More detailed models like Barry Boehm’s COCOMO are not metrics per se, but complete systems that consider all known factors that affect software costs.

They can be fined tuned to an organization to make very accurate estimates for the organization.

However, they are very laborious to use.

This Talk

This talk, based on Eli Kantorowitz's paper "Algorithm Simplification through Object Orientation", ***Software—Practice and Experience*****, February, 1997, has two purposes,**

1. **showing how object orientation (OO) can indeed simplify programs in a way that makes them demonstrably much easier to develop and maintain, and**

This Talk, Cont’d

2. introducing the first real *programming* complexity measure, called *logical complexity*, that really measures development and maintenance difficulty.

These are done by examination of a specific example of program development and enhancement that was helped by use of OO.

Logical Complexity Refined

Logical complexity is itself two measures, *development complexity* and *extension complexity*, measuring two different phases of the lifecycle.

The Breakthrough

The logical complexity measure is a real breakthrough, but unfortunately, the measure is a function of problem-specific data and not of the code structure, as are the extant measures.

Thus, the particular measure used for this program measure cannot be used for another program.

The Breakthrough, Cont'd

However, the idea of finding such a measure can be used anywhere; it just requires creativity.

Given that code-structure based measures do not work very well, maybe problem-specific complexity functions are the way to go.

I have told you all this now, so that you will be watching for the idea.

Old vs. New Design Methods

Old Method:

- **Design algorithm.**
- **Implement algorithm top-down (TD) in any language.**

It is believed that the algorithm can be designed independently of the programming language.

Old vs. New Methods, Cont’d

New Method:

- **Identify objects and their operations from problem description.**
- **Write OO program using these objects.**

OO programs mimic the real-world situation that they model and are thus easier to validate.

Puzzle

Observation in case study:

A program developed by OO reasoning was simpler to develop, understand, and maintain than one developed by the traditional TD method for the same application even though both programs had the same time and space complexity.

The OO program has a significantly smaller logical complexity than the TD program.

Case: CAD System

CAD system aided the design of a structure with about 10^5 parts.

The system was 20 years old, complicated, buggy, difficult to extend, and even more difficult to extend with each extension.

Case: CAD System, Cont'd

Thus, it was decided to re-engineer the system, to build a new one with the same functionality, that would be less complicated, less buggy, easier to extend, and maybe even not more difficult to extend with each extension.

The old system was built in the traditional, TD manner, and the new system was built in an OO manner.

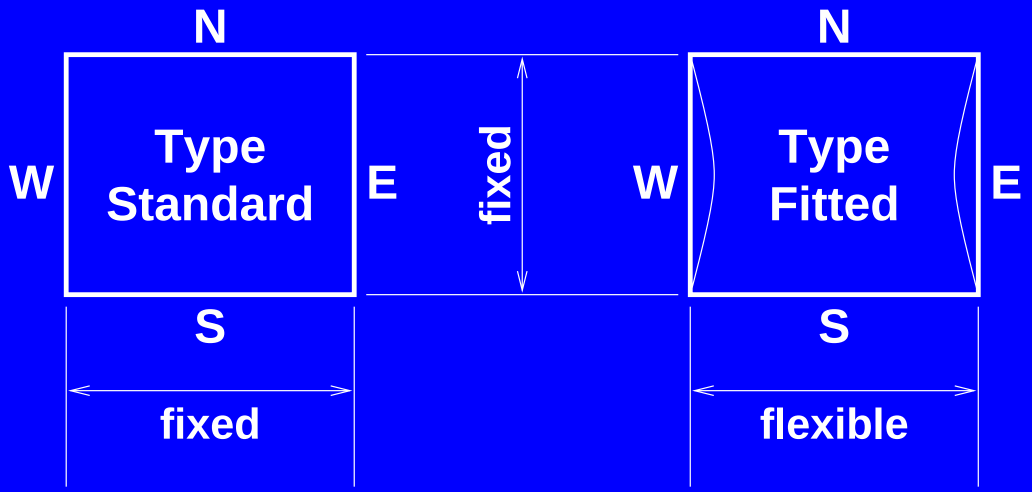
A 2D Simplification

The system handled parts in 3D.

The description below is a 2D simplification to allow easier visualization of the issues.

The complexity analyses will be given also for the 3D case, as an obvious analogy to the 2D case.

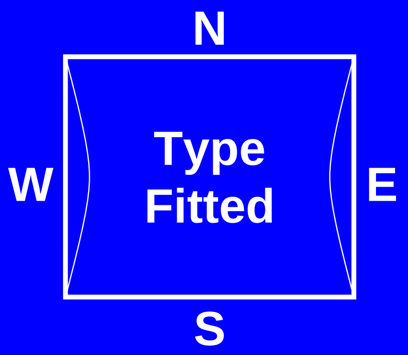
Rectangular Parts

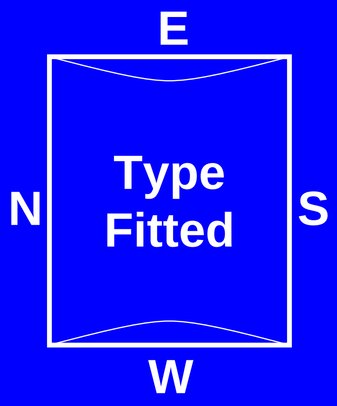


Standard and Fitted Parts.

Rotation of Parts

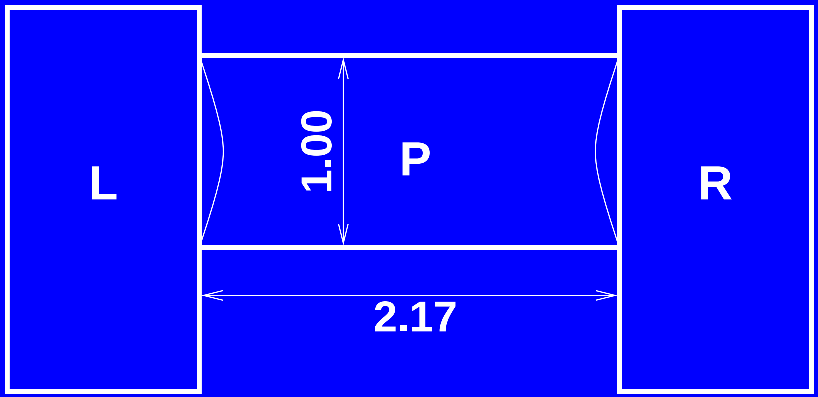
Any part can be rotated any multiple of 90°.





Therefore, it is necessary to talk about fitted faces (E and W) of fitted parts and standard faces (N and S) of fitted parts.

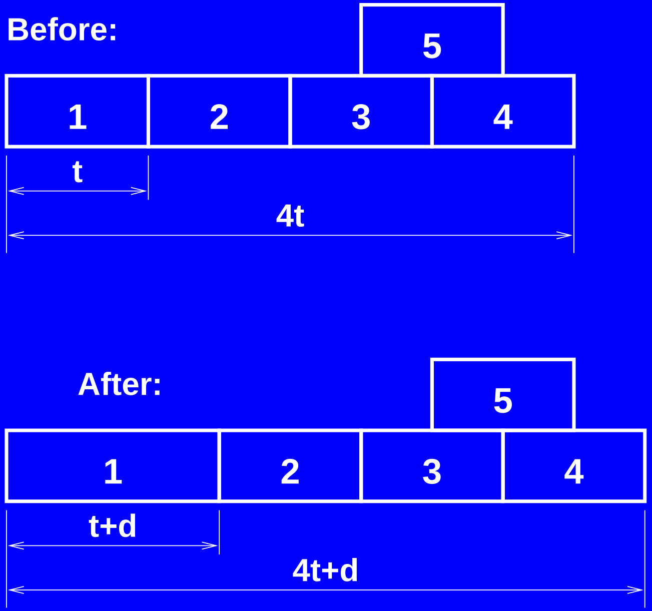
Fitting Parts Together



Fitted part between two standard parts.

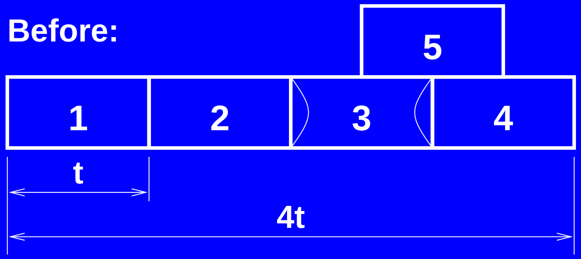
Change Propagation

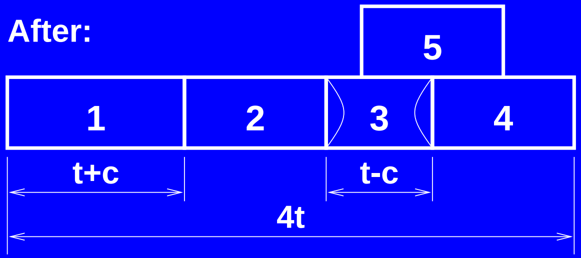
Through standard part or through N or S face of fitted part.



Change Propagation, Cont'd

Through E or W face of fitted part.





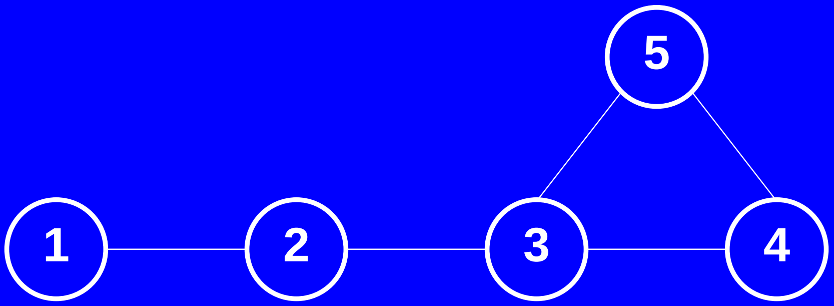
Propagation Rules

- **When a standard face of a part P is pushed a distance d, part P is moved d in the push direction. The width of P along the push direction is not changed.**
- **When a fitted face (E or W) of a fitted part P is pushed a distance c, part P is not moved. The width of P along the push direction is reduced by c.**

Structure of Old Program

Designed TD.

Think of network of connected parts, e.g., for the propagation examples above.



Structure, Cont’d

The main program builds the network.

When it is asked to apply a change to a part, it follows the propagation rules.

Structure, Cont’d

Thus, it must deal with the interface between two parts.

There is one routine for each such interface, that can be called by the main program.

Number of Interface Routines

There are

- 2 part types
- 4 faces per part
- for a total of $2 \times 4 = 8$ face types
- for a total of $8 \times 8 = 64$ interfaces

Number of Routines, Cont'd

More generally in the 2D case

$$N_{face\_types} = N_{faces} \times N_{types} = 4 \times N_{types} = 8$$

$$N_{interfaces} = N^2_{face\_types} = 4 \times N^2_{types} = 64$$

Behavior of 64 Interfaces

	S-N	S-E	S-S	S-W	F-N	F-E	F-S	F-W
S-N	m	m	m	m	m	a	m	a
S-E	m	m	m	m	m	a	m	a
S-S	m	m	m	m	m	a	m	a
S-W	m	m	m	m	m	a	m	a
F-N	m	m	m	m	m	a	m	a
F-E	a	a	a	a	a	e	a	e
F-S	m	m	m	m	m	a	m	a
F-W	a	a	a	a	a	e	a	e

a=absorb, m=move, e=error

Behavior of 64 Interfaces, Cont’d

This leads to 64 routines, one for each possible interface.

The code for each routine is determined by the rules for the interface represented by the routine.

There is much code in common in these routines, but no two are exactly the same.

Behavior of 64 Interfaces, Cont'd

Keeping track of these minor differences is a real headache.

Putting the common code into lower level subroutines helps a bit in that what is common is stands out as a larger fraction of the code.

However, the minor differences are a real pain to keep track of.

3D Analysis

- 9 types of 3D parts
- 6 faces per part type
- $N_{face\_types} = N_{types} \times N_{faces} = 9 \times 6 = 54$
- $N_{interfaces} = N_{face\_types}^2 = 54 \times 54 = 2916$
- 2916 subroutines with lots of duplicated code

Development Complexity

The development complexity (of the logical complexity) in 2D and 3D cases of the TD program is:

$$O\,(N_{types}^2)$$

Difficulties

2916 subroutines with lots of duplicated code!

Difficult to program them correctly

Difficult to debug them

Difficult to manage them

Difficult to maintain them

Difficult to extend the program

Difficult to extend

Adding a new 2D part type with 4 faces

- **3 part types**
- **4 faces per part**
- **for a total of $3 \times 4 = 12$ face types**
- **for a total of $12 \times 12 = 144$ interfaces**
- **80 more than before**

Difficult to extend, Cont'd

There are 80 more routines with lots of code in common with the original 64, but no two are exactly the same, for even more headache.

More generally,

$$(4\times(N_{types}+1))^2-(4\times N_{types})^2=32\times N_{types}+16$$

3D Analysis

Adding a new part type with 6 faces adds

$$(6\times(N_{types}+1))^2-(6\times N_{types})^2=72\times N_{types}+36$$

new interfaces to be managed.

If we had 9 parts to begin with, we now have 684 new interfaces to be managed.

If we had 10 parts to begin with, we now have 756 new interfaces to be managed.

etc.

3D Analysis, Cont'd

As N_{types} grows, extending the software becomes very laborious and very error-prone because of the minute differences in the face of large amounts of common code.

Sometimes N_{types} increases considerably.

Extension Complexity

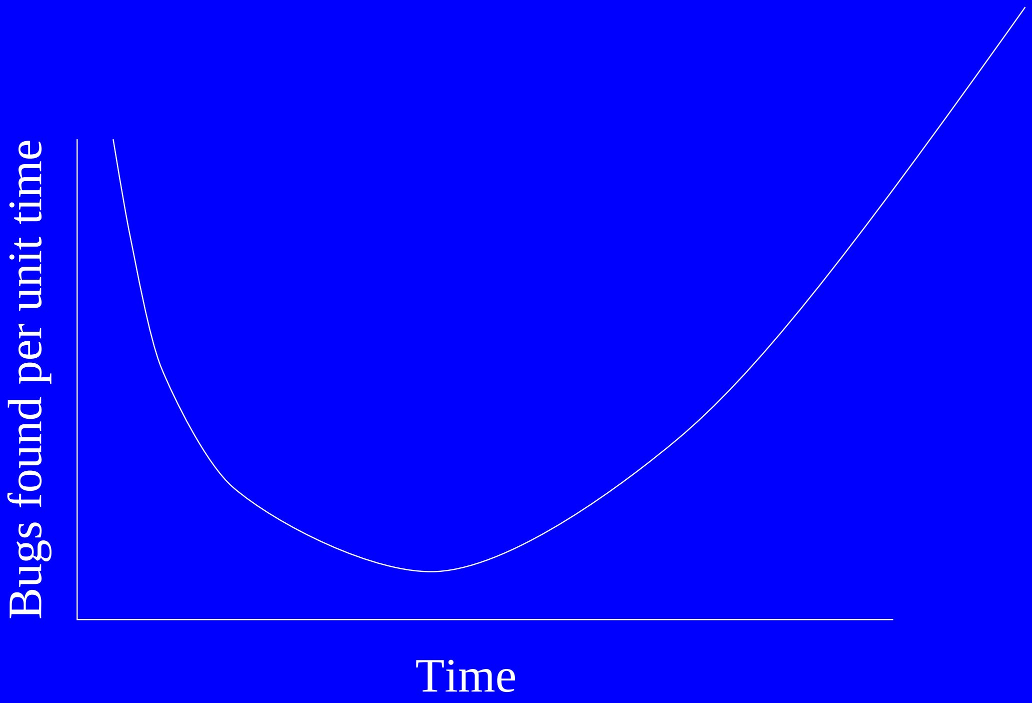
The extension complexity (of the logical complexity) in 2D and 3D cases of the TD program is:

$$O\,(N_{types})$$

And the multiplicative constants are large.

Belady-Lehman Law

This is a classic example of the situation described by the famous Belady-Lehman graph!



Note that this program is not BAD.

The original structure is good. It does decompose the problem in such a way that deciding what each routine does is very systematic.

It could have been a *lot* worse, a whole lot.

But at the point at which the number of bugs per release seemed to be growing, it was time to re-engineer the program.

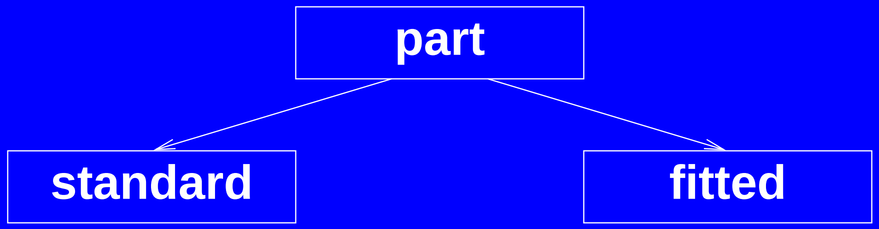
This re-engineering was carried out in an OO manner.

Object Identification

Identify objects from real world

```
class part;

class standard public part;
class fitted public part;
```



Operation Identification

Identity operations on part, inherited by subclasses.

```
push_on_N_face(float distance);
push_on_E_face(float distance);
push_on_W_face(float distance);
push_on_S_face(float distance);
```

The rules determine the code for each body.

Actually, the operations are virtual in part and are defined in subclasses.

Main Program

- **builds graph of part connections with an object of correct type at each node**
- **pushes on the correct face of the changed part**
- **propagation**
  - **a push on a fitted face is absorbed, the containing part is shortened and no propagation occurs**
  - **a push on a standard face causes a push to the neighbor on the opposite face (sending a message)**

Number of Operations in Part Class

There are

- **2 types**
- **4 faces**
- **$2 \times 4 = 8$ face-type operations**

More generally,

$$N_{operations} = N_{types} \times N_{faces} = 2 \times 4 = 8$$

3D Analysis

$$N_{operations} = N_{types} \times N_{faces} = 9 \times 6 = 54$$

Development Complexity

The development complexity (of the logical complexity) in 2D and 3D cases of the OO program is:

$$O(N_{types})$$

Adding New Type

If we add a new type with four faces, we must add one more subclass to part and write its four operations, adding 4 new operations for a total of $3 \times 4 = 12$ face-type operations.

$$(4 \times (N_{types} + 1)) - (4 \times N_{types}) = 4$$

3D Analysis

Adding a new part type with 6 faces adds

$$(6\times(N_{types}+1))-(6\times N_{types})=6$$

new operations to program, considerably smaller than 684, and the number added per new type does not grow!

Extension Complexity

The extension complexity (of the logical complexity) in 2D and 3D cases of the OO program is:

$$O\,(1)$$

Comparisons

Number of routines to program, the development complexity:

- **Old:** $O(N^2_{face\_types})$
- **New:** $O(N_{face\_types})$

Note that $O(N_{face\_types}) = O(N_{types})$

Comparisons, Cont'd

Number of routines to program for each new part type, the extension complexity:

- **Old:** $O(N_{face\_types})$
- **New:** $O(1)$

Qualitative analysis

Observe that the time and space complexities of the new and old programs are the same.

Indeed, the irony is that (looking at the 2D case) the code of each of the 64 interface routines in the old program is the merger of the code of two push routines in the new program.

Qualitative analysis, Cont’d

In the old program, the merger is done at coding time.

In the new program, the merger is done at run time.

In Retrospect

We could have seen this decomposition of the problem back then and could have used macros equivalent to the bodies of the new program push routines to build the 64 interface routines.

We did not, probably because we were just not thinking that way back then.

Therefore, programming paradigms and language features do make a difference.

In Retrospect, Cont’d

They do suggest different ways to view a problem, which in turn suggests different ways to decompose the code into modules.

As Parnas observed with the KWIC example, all of these different decompositions may end up as the same run-time code with the same performance.

Logical Complexity

We call this complexity of programming the logical complexity.

This is a true measure of programming effort.

Alas, it is very problem-specific, and is not applicable to any other programming problem, especially one that does not have any notion of part types and faces, etc.

Logical Complexity, Cont'd

Therefore, for each problem, it is necessary to identify from scratch, the problem-specific variables that are parameters of its logical complexity.

Maybe this is the best that can be done, if one wants something more accurate than the existing measures based on code-structure.

Logical Complexity, Cont'd

Note that this problem-specific logical complexity is very accurate in its prediction of development and maintenance effort.

This is more than can be said of the extant, code-structure-based metrics.

Moreover, it requires a lot less effort to calculate logical complexity than to use the more accurate systems like COCOMO.

Generalizing

Kantorowitz suggests finding

- **some problem-dependent data *P* that changes linearly with each change to the problem, and**
- **some problem-dependent data *C* that changes linearly with the amount of code to be written anew or changed whenever *P* changes.**

Generalizing, Cont'd

The function calculating *C* from *P* is developmental complexity.

The function calculating Δ*C* from Δ*P* is the extension complexity.