

# **Controlling Complexity in Enterprise Architectures**

## **Mathematical Foundations Part II of III**



**A Briefing Paper by Roger Sessions  
ObjectWatch, Inc.**

## Introduction

Complexity is the insidious enemy of enterprise architectures. It is critical to have a methodology that recognizes and manages this enemy effectively. Unless complexity is managed, the chances of an enterprise architecture ever delivering business value are small and the likelihood of costly failure is high.

SIP is a new methodology for enterprise architectures. SIP stands for *Simple Iterative Partitions*. SIP is designed specifically to control complexity.

While many methodologies claim to address complexity, these claims are not testable because the methodologies lack models for complexity. While these methodologies produce results that can be described as enterprise architectures, there is no way to know if the resulting architectures are “good” or “bad”, or if better enterprise architectures are possible.

SIP is different. SIP is firmly grounded in a mathematical model for complexity and is composed of logical rules for simplification of architectures. Resulting architectures can be tested according to these rules to see if they are “good” or “bad”, and/or if they can be improved.

Of course, we must start by agreeing on definitions of “good” and “bad”. Our premise is that “good” is most closely associated with the characteristic of *simplicity*. Of two architectures that both meet business need, the simpler architecture is the better one.

The cost differentials between simple and complex architectures can be amazing— often a full order of magnitude or more. This means, for example, that a system that might have a lifetime cost of, say, \$100 million using traditional approaches could save \$90 million (or more) using SIP approaches to complexity management. Sounds almost unbelievable, right? And yet, it is true. With reduced complexity comes faster implementations and easier maintenance. SIP delivers in every phase of an IT system lifecycle.

What is SIP’s mathematical model for complexity? How can we test an enterprise architecture for optimal design against this model? This is the subject of this briefing paper.

## Comparing Two Programs

Consider a simple IT system that manages penny tosses. A person will toss a penny in a penny-sensor; the sensor will “read” the penny to see if it lands heads or tails; and a software system will show an appropriate message on a monitor. Overall, the configuration is shown in Figure 1.

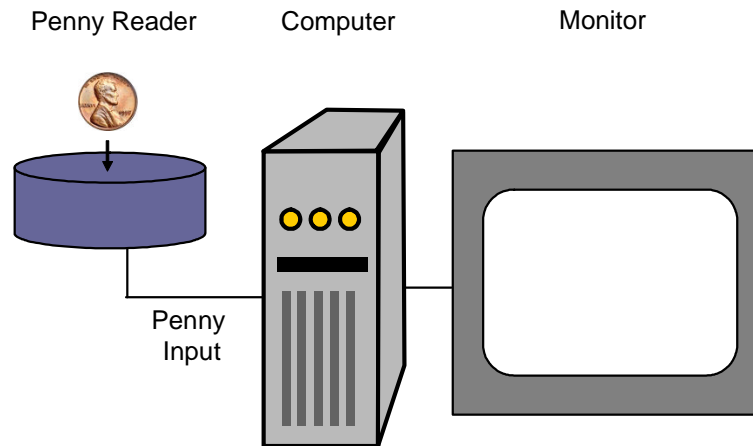


Figure 1. Penny Toss Reader

The software system that runs in the computer might look something like the following:

```
values (heads, tails) penny;  
penny = read (penny_sensor);  
if (penny = heads) message ("Penny is heads");  
if (penny = tails) message ("Penny is tails");  
end;
```

Suppose that we wanted to prove that this program works for every possible state in which it might find itself. Assuming the hardware works as advertised, how many tests would we need to run to convince ourselves that the program works? We would need to run two tests.

The first test would be to drop a penny in the penny toss reader with the heads up. If the message came up, "Penny is heads", then the test would pass. If the message came up, "Penny is tails" then the test would fail.

The second test would be to drop a penny in the reader tails up. If the message came up, "Penny is tails", then the test would pass. If the message came up, "Penny is heads" then the test would fail.

Neither test by itself is sufficient. We need to run both tests before we know the program works in all possible cases.

Now, let's consider a system that monitors penny and dime tosses. The overall configuration for this system is shown in Figure 2.

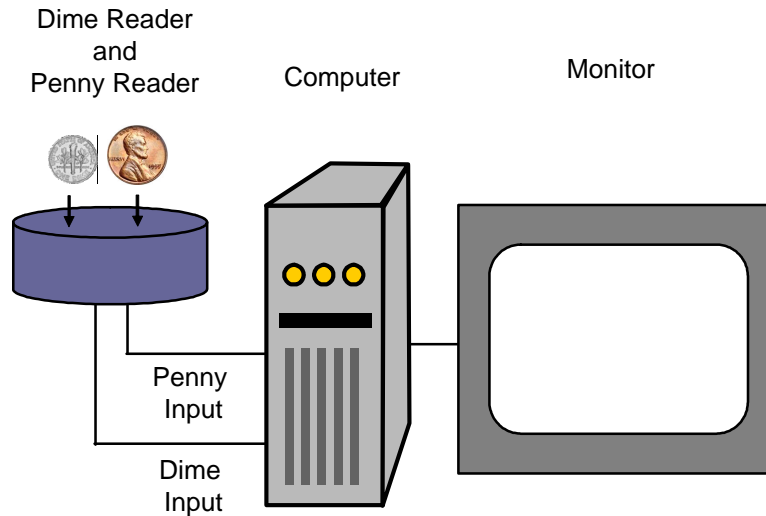


Figure 2. Penny and Dime Toss Reader

The software system in the second system might look something like the following:

```

values (heads, tails) penny, dime;
penny = read (penny_sensor);
dime = read (dime_sensor);
if (penny = heads and dime = heads) message ("Penny is heads Dime is heads");
if (penny = heads and dime = tails ) message ("Penny is heads Dime is tails");
if (penny = tails and dime = tails) message ("Penny is tails Dime is tails");
if (penny = tails and dime = heads) message ("Penny is tails Dime is heads");
end;

```

Which program is more complex - the penny-only program or the penny-dime program?

I think most of us would agree the penny-dime program is significantly more complex than the penny-only program. There are more lines of code that could go wrong and each line is more complex than the analogous lines in the penny-only program.

Were we to review each "if" statement in the program, we would have twice as many statements to examine in the penny-dime program. Based on required review time, we could say that the penny-dime program is twice as complex as the penny-only program.

How many states can each of the programs find itself in? The penny-only program can find itself in one of two states. The penny-dime program can find itself in one of four states. Because the penny-dime program has twice as many significant states as the penny-only program, it requires twice as many tests to prove that it works in all possible cases.

This analysis strongly suggests that there is a direct relationship between the complexity of a program and the number of states in which a program can find itself.

The number of states in which a program can find itself is a function of both the number of variables in the program and the number of significant states that each variable can take. As either of these numbers increases, so does the number of program states, and, therefore, the program complexity. In the penny-dime program, we doubled the number of variables and doubled the complexity.

If  $V$  is the number of variables in a program and  $S$  is the number of significant states each variable can take, then the total number of states for the program as a whole ( $T$ ) is given by

$$T = S^V$$

If a program has 2 variables, each of which can take 6 states, then the total number of states is 36 ( $6^2$ ). This program will be 6 times more complex than a program with only one variable that takes six states. This latter program has only 6 significant states ( $6^1$ ).

As one adds variables in the program, the number of possible states, and thus complexity, increases exponentially. Table 1 shows the number of states of a program as one adds six-state variables.

Variables	Program States
1	6
2	36
3	216
4	1,296
5	7,776
6	46,656
7	279,936
8	1,679,616
9	10,077,696
10	60,466,176
11	362,797,056
12	2,176,782,336

Table 1. Program States as Function of Variable Number ( $S$  is 6)

Table 1 seems to show that any time you have twelve six-stated variables, you have a very complex system. But that is not necessarily so. It is only true when the variables are all thrown together into a single program. Suppose we could separate the variables, so that instead of one large twelve-variable system, we have two smaller six-variable systems.

Now, rather than have one program of 2 billion states, we have two programs of 46,000 states each, for a total of 92,000 states. Since complexity is directly related to the number of states, **we have just reduced the complexity of our overall system by more than 99%**. And, amazingly, we have done so without removing a single variable!

## Partitioning

Mathematically, we have just taken a single set of 12 variables and split it into two subsets of 6 variables each. Moreover, these 2 subsets represent a mathematical *partition* of the original set.

A partition is a concept that comes from *set theory*. A partition is a set of subsets that divide up a larger set such that all points in the larger set live in one, and only one, of the subsets.

Partitioning sounds simple, doesn't it? Just split up the 12 variables into two subsets that form a partition! But these variables are used by program logic. How do you know which variables can be separated from which other variables?

Mathematically, partitions are closely related to another concept called *equivalence relations*. An equivalence relation ( $E$ ) is a binary (true/false) relation over some universe of elements such that for any elements — say,  $a$ ,  $b$ , and  $c$  — the following three properties are always true:

- $E(a, a)$  is always true — known as the property of *reflexivity*
- $E(a, b)$  always implies  $E(b, a)$  — known as the property of *symmetry*
- $E(a, b)$  and  $E(b, c)$  always implies  $E(a, c)$  — known as the property of *transitivity*

Consider a small store. We can think of all of the items in a store as being a particular universe. The relation *costs the same as* is an equivalence relation. Let's say the store currently stocks the following ten items:

cereal	\$1.00	flour	\$2.00
notebook	\$2.00	newspaper	\$ .50
pencil	\$ .50	soda	\$1.00
pen	\$1.00	cup	\$2.00
candy	\$ .75	knife	\$5.00

Here are some *true* equivalence relations:

- costs the same as* (cereal, pen)
- costs the same as* (notebook, flour)

Here are some *false* equivalence relations:

- costs the same as* (cereal, pencil)
- costs the same as* (candy, newspaper)

Notice that the three equivalence relation properties hold true for the *cost the same as* relation:

- costs the same as*(cereal, cereal) = reflexivity
- costs the same as* (cereal, pen) implies *costs the same as* (pen, cereal) = symmetry

*costs the same as* (cereal, pen) and *costs the same as* (pen, soda) implies  
*costs the same as* (cereal, soda) = transitivity

These three properties, of course, are true for all elements of the universe. I have just chosen some representative elements to illustrate the properties.

What makes equivalence relations interesting from the perspective of partitioning is that equivalence relations can be used to manufacture unique partitions. For example, if we create subsets of our store items for which the *costs the same as* equivalence relation is true, we get the following subsets:

Subset 1: cereal, pen, soda  
Subset 2: notebook, flour, cup  
Subset 3: pencil, newspaper  
Subset 4: candy  
Subset 5: knife

Since every element of our original universe lives in one, and only one subset, these five subsets form a valid partition. Thus we can use *equivalence relations* to manufacture *partitions*.

Notice an interesting fact: while there are many valid partitions for this particular universe, there is one, and only one, partition that results from the application of this particular equivalence relation.

Now, let's go back to our earlier problem—how to split up our program of 12 variables into 2 (or more) programs of, at most, 6 variables each such that the new programs partition the original variable space. Remember, the payoff for this is a 99%+ reduction in complexity.

We can do this if we can find an equivalence relation on the variables. Any equivalence relation will do as far as creating a valid partition, but we would like one that specifically preserves logical dependencies between the variables. This would have the useful side benefit that we could still expect our programs to function correctly at the end of the exercise!

That brings us to SIP. Simple Iterative Partitions (SIP) seeks an equivalence relation not on the variables themselves, but, rather, on the *functionality* of the overall system. The reasoning is that functionality indirectly controls variables. Adding functionality to a system effectively adds variables to a system. Partitioning functionality effectively partitions the underlying related variables.

So, SIP focuses not on equivalence relations on variables (which are really an implementation detail that live at a lower level than enterprise architectures) *but on equivalence relations on functionality* (which live squarely in the domain of enterprise architecture.)

The equivalence relation that is of most interest to us is *synergy*. Two functions are *synergistic* when each requires the other to be effective. The inverse of synergistic is *autonomous*.

Whereas *synergistic* is an equivalence relation and therefore can be used to define a valid partition, *autonomous*—its inverse— is not an equivalence relation. The inverse of an equivalence relation is never an equivalence relation. You can see this for yourself if you think about the *doesn't cost the same as* relation, which is the inverse of the *costs the same as* equivalence relation. Transitivity, for example, is not a property of *doesn't cost the same as*.

While the inverse of an equivalence relation is never itself an equivalence relation (and therefore cannot be used as a basis for partitioning), the fact that it is an inverse of an equivalence relation gives it some interesting mathematical properties. In the area of enterprise architecture, this is especially true for the inverse equivalence relation *autonomous*. Regrettably this discussion would take us beyond the scope of the present paper, so I will simply leave it with you as food for thought.

Let's consider some examples of the universe of functionality that we might encounter in creating an enterprise architecture for a retail operation. Some possibilities are:

- *calculate total cost*
- *calculate change*
- *charge credit card*
- *remove from inventory*
- *report on current inventory*

Applying the *synergy* test to this universe, we can see that *calculating total cost* is synergistic with *calculating change*. It's hard to imagine either without the other.

At first glance, it might appear that *charge credit card* is also synergistic with *calculate total cost*, since you can't charge a credit card without knowing the total cost. However, *synergy* is a two-way street—it requires *mutual* dependency (the property of *symmetry*). It is possible to imagine situations where you would want to calculate the total cost without charging credit cards (in a cash-only transaction, for example). Therefore *charge credit card* is not considered synergistic with *calculate change*. Rather, they are considered *autonomous*.

It is easy to show that synergy, as defined here, is an equivalence relation, since it satisfies the three properties of equivalence relations (reflexivity, symmetry, and transitivity). Therefore, it, like all equivalence relations, also defines a *partition* of our universe. And further, it defines not only a partition, but a *unique* partition. There is only one partition that can result from applying this (or any) equivalence relation.

But *synergy* specifically has features that go beyond even those of your run-of-the-mill equivalence relation. Not only does *synergy* define a *unique* partition on our universe (as would any equivalence relation), it defines an *optimal* partition on our universe with respect to managing complexity. Any smaller partitions and we would run into functional dependencies. Any larger partitions and we would have unnecessary complexity.

In other words, applying the synergy equivalence relation against a universe of enterprise architectural functionality yields the *best possible partition* of that universe based on the criteria of complexity. And controlling complexity, you may recall, is job number one.

## Beyond Partitions

Believe it or not, partitioning is only one of the ways SIP manages complexity. It also uses consolidation, elimination, and outsourcing to further reduce the complexity of individual partitions. It also uses iteration to rollout partitions in the best possible sequence. However this is enough math for today and, partitioning is the most important tool in the SIP tool chest for managing complexity.

## Summary

Let me briefly restate the important points of this briefing paper:

- System complexity is public enemy number one for enterprise architectures.
- System complexity is directly related to the number of possible states of that system. Control the number of states of a system and you control that system's complexity.
- The best way to control the number of states is to use mathematical partitions. Even a small number of partitions dramatically reduces overall complexity.
- The best way to create partitions is through applications of equivalence relations.
- The best equivalence relation, with respect to managing complexity, is the relation *synergistic*.
- The inverse of *synergistic* - *autonomous* - while not an equivalence relation, is plenty interesting in its own right.

## Conclusion

Nobody would think of sending a rocket to the moon without testing the planned trajectory against mathematical models for gravity and planetary motion. Nobody would think of building a bridge without testing the architecture against mathematical models for stress and load. However, we commonly implement large, expensive IT projects without any idea whether they are based on a sound architecture or not. Many of these projects fail - often at great cost. Projects fail so frequently because we have no model that defines "good" enterprise architectures.

That is why the SIP model for complexity is so important. It gives us a process for validating an architecture in the early design phases, when changes can be made more easily. It gives us a logical approach for controlling complexity - the single biggest reason that large IT projects fail.

The reality is that all enterprise architectures *will* be validated. On that you have no choice. What you do have a choice on is *when* that validation will occur: sooner or later. With SIP, this validation can occur early in the design process when corrections can be made more easily and

cheaply. Without SIP, this validation will likely occur much later, after implementation, when corrections are made only at great cost and risk.

The answer to complexity is simple. It is SIP.

## More on SIP

SIP was invented by Roger Session. Roger Sessions is the CTO of ObjectWatch. He has written six books (including *Software Fortresses; Modeling Enterprise Architectures*) and many articles. He is on the Board of Directors of the International Association of Software Architects (IASA), Editor-in-Chief of *Perspectives of the International Association of Software Architects*, and a Microsoft™ recognized MVP in Enterprise Architecture. He holds multiple patents in both software and enterprise architecture process. An accomplished key note speaker, Mr. Sessions has given talks at more than 100 conferences around the world covering a wide range of topics in Enterprise Architecture.

For more information on how SIP can help you manage the complexity of your IT projects, write [information@objectwatch.com](mailto:information@objectwatch.com).

## Legal Notices

The SIP methodology is protected by pending patents. SIP and Simple Iterative Partitions are trademarks of ObjectWatch, Inc. ObjectWatch is a registered trademark of ObjectWatch, Inc. This briefing paper is copyright© 2007 by ObjectWatch, Inc.