What is Killing the

Intelligence Dinosaurs?

March 2004

John Fairweather, President

MitoSystems, Inc.
3205 Ocean Park Blvd., Suite 180
Santa Monica, CA 90405
Tel: (310) 581-3600
FAX: (310) 581-3777

Table of Contents

List of Tables

List of Figures

1. Introduction

This white paper discusses problems faced by intelligence organizations in adapting to the rapidly changing and hugely complex information environment that exists in the world today, the difficulties inherent in monitoring this information ocean, and extracting indicators and warnings from it to provide early notification to decision makers of existing or impending terrorist threats. Central to understanding the reasons for past intelligence failures and exponentiating intelligence system costs (and failures) is a full understanding of the decision cycle or Observe Orient Decide Act ("OODA") loop teachings of Colonel John Boyd (1927 – 1997), considered by some to be one of the foremost thinkers in military strategy of all time. Unfortunately, much of Boyd's teachings continue to be ignored today, and it is the failure to fully and systematically embrace these teachings, which lies at the root of many of the problems faced by intelligence organizations today. The central point is that change is a pervasive feature of the information world, and that it is the failure to systematically address change and its impact that results in many of the dramatic failures that we see today. These failures will continue in the future if we do not learn from our mistakes and radically alter the way we collect, analyze, and disseminate information.

This paper describes the infrastructural and technical impact of an OODA loop-based mind-set (or the lack thereof) and presents many reasons why failure to adopt this mind-set has contributed to intelligence failures in the past, and continues to do so today. The technical issues that lead to the failure of complex intelligence systems are examined in detail and illustrated in depth using a specific example. Finally, the paper discusses the advantages of using the Mitopia® architecture, developed over the last 14 years specifically, to address these problems in the intelligence field, and illustrates the significant benefits to be achieved by moving to such an architecture. The paper presents concrete parallel findings for the chosen example when addressed using a Mitopia®-based approach, and shows that in using such an approach it is possible to address the problem given, whereas using the standard approaches popular today, it is not.

The author has spent over 15 years developing the field of adaptive intelligence architectures, specifically in the design and implementation of the architecture that has become Mitopia®. It is the perspective and experience gleaned from this singular focus, and the frustrations experienced in seeing others continue to pursue outmoded and technically bankrupt approaches, that lead to the writing of this paper. It is hoped that those in charge of plotting our future systems' course will learn from, and incorporate, the points raised in this paper, whether by use of Mitopia™ or otherwise. Failure to do so, in the author's opinion, would constitute an abrogation of responsibility to our collective futures.

2. What is the Problem?

[1] The problem in a nutshell is failure to adapt.

The world has changed; the intelligence threats we face are no longer few and large, slow moving, and relatively static, they are small, many and fecund; they are diverse and adapting or evolving at a ferocious rate. Our failure to adapt is both a technical problem as well as an organizational and infrastructure problem, and it comes from a complete failure on the part of our existing technologies and our existing intelligence organizations to consider the impact of decision cycles or OODA loops in any of their doings. These organizational failings are largely the result of ignorance and complacence. As a result, the government lags by decades behind the commercial world. Companies failing to adapt to a competitive business-cycle approach are now becoming extinct. They have been selected out for failure to adapt. As far as our technical failings to facilitate an OODA loop-based framework, industry is in the same boat as the government here. Simply put, the systems to enable this approach simply never existed in order to be adopted.

2.1 The OODA Loop

[1] Modern competitive and business intelligence cycles are now based on some derivative of the Boyd Cycle (or OODA loop). This cycle was developed by Colonel John Boyd as a result of his studies (and experience) of air-to-air combat in the Korean War. What Boyd discovered was that the main factors that enabled U.S. pilots to consistently win dogfights were firstly that the F-86 fighter aircraft's canopy was larger than that of the opposing Mig-15's, thus giving a greater field of vision, and secondly, that although the F-86 aircraft was larger and slower, it was more maneuverable (higher roll-rate), thus allowing US pilots to make more frequent adjustments. Boyd was later largely responsible for the design of the F-15 canopy and perhaps more than anyone else, contributed to development and deployment of the F-16. Boyd also helped create the Fighter Weapons School at Nellis AFB, Nevada. The result of formalizing and abstracting Boyd's insight became a fundamental part of air-force fighter tactics, and was later embraced fully by the Marines. The bulk of the defense establishment, however, has ignored or is unaware of his teachings. In the intelligence community, his work is virtually unknown.

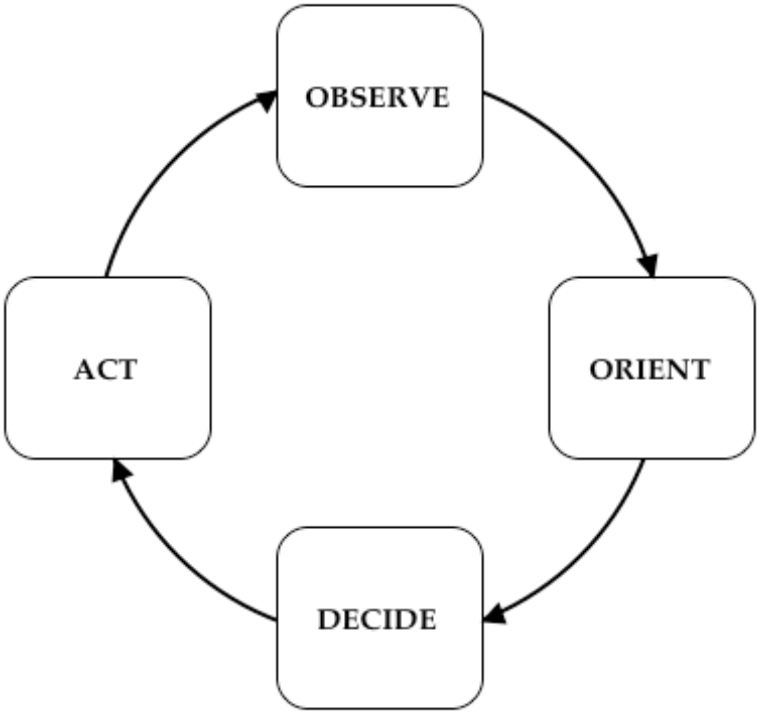


Figure 2.1 The Boyd Cycle (OODA Loop)

[2] The central idea behind the OODA loop is that all thinking entities are executing OODA loops of their own (consciously or otherwise). The key to success in any conflict or competition is therefore either:

a) Being able to cycle around the loop faster than your opponent
b) Disrupt the opponents OODA loop to cause him to slow down or make mistakes
c) Alter the tempo and rhythms of your own loop so that the opponent cannot keep up with you

[3] For a full description of the OODA loop and how it ties in with the intelligence problem, as well as a complete bibliography in this area, see the paper **"Avoiding Information Overload Through the Understanding of OODA Loops, A Cognitive Hierarchy and Object-Oriented Analysis and Design"** by Dr. R.J. Curts, CDR, USN (Ret.), and Dr. D.E. Campbell, LCDR, USNR-R(Ret.). This paper can be downloaded from www.belisarius.com. This site deals with business intelligence, and is heavily focused on the work of Boyd. While this author is not in complete agreement with the paper's assertion that object-oriented techniques provide a practical approach to addressing the issue, the paper does effectively describe the need for a ground-up approach, and a consistent method for representing and storing data.

2.2 Infrastructure Problems

[1] The traditional intelligence cycle is depicted in the diagram below. In this model, the intelligence consumers make known their needs for information via requests that are passed to the organization that assigns priorities to information requirements. Determination of priorities leads to tasking, which results in the various collection mechanisms or agencies taking steps to gather the raw information necessary to pass on to the analysts. After performing whatever analyses best fit the problem domain, the analysts prepare reports, which are then reviewed and coordinated and finally disseminated back to the original intelligence consumer.

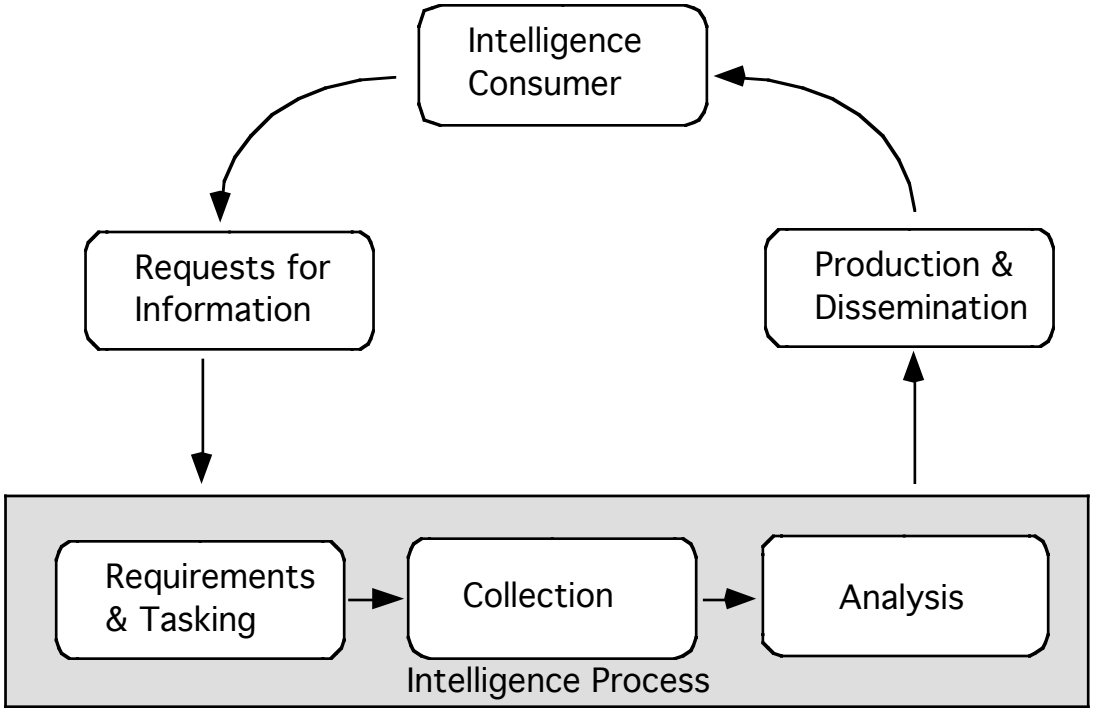


Figure 2.2 The Traditional Intelligence Cycle

[2] The cycle described above represents the best thinking on how intelligence should work from the 1940's and 1950's. The cycle is still utilized today by the government intelligence community. In today's fast moving and information-rich environment, such a cycle is unfortunately inadequate to the task of tracking the complexities of unfolding world events. A full description of the problems with such a cycle is beyond the scope of this document, however, the basic problems can be summarized as follows:

a) The cycle is too slow. Indeed it is not clear that it is a cycle at all, since most requests result in just a single iteration. The existence of various organizations (bureaucracies) in the cycle combined with the time taken for information to pass through the bureaucratic interfaces in the loop mean that the cycle cannot keep up with evolving events.

b) Because it is essentially command driven, the cycle only allows looking into questions that the intelligence consumer already 'knows' to ask. As discussed previously, the reality is that the cycle must support the discovery of things you didn't even know were important. The September 11th attacks provide a perfect example. This top-down approach may have suited a situation where the enemy was known and stable (i.e., U.S.S.R.), but it does not deal well with today's world where enemies are small, distributed, loosely coupled, change constantly, and can have impacts disproportionate to their size. The intelligence consumer cannot anticipate all possible threats and task the complete cycle to investigate each.

c) The lack of feedback in the cycle between the consumer and the analyst, combined with the inability of the consumer to directly access and examine the backup material leading to analytical conclusions, tends to create a situation where the final product may not meet the consumer's requirements, and thus redundant iterations through the cycle with corresponding increases in time and cost are required.

[3] If we explore the reasons behind the community's failure to move to a modern decision cycle based on the OODA loop (or a derivative) we see that is primarily due to a failure to adapt to the new threat environment posed by a post-cold-war world. Specifically, the fact that the enemy is no longer the U.S.S.R. but a diverse, loosely coupled, and rapidly adapting amalgam of terrorist organizations and other assorted whackos. Only a fraction of the intelligence problem now relates to threats posed by sovereign states. In the good old days of the Cold War, the opposing intelligence forces (viewed in OODA loop terms) looked something like that depicted below:

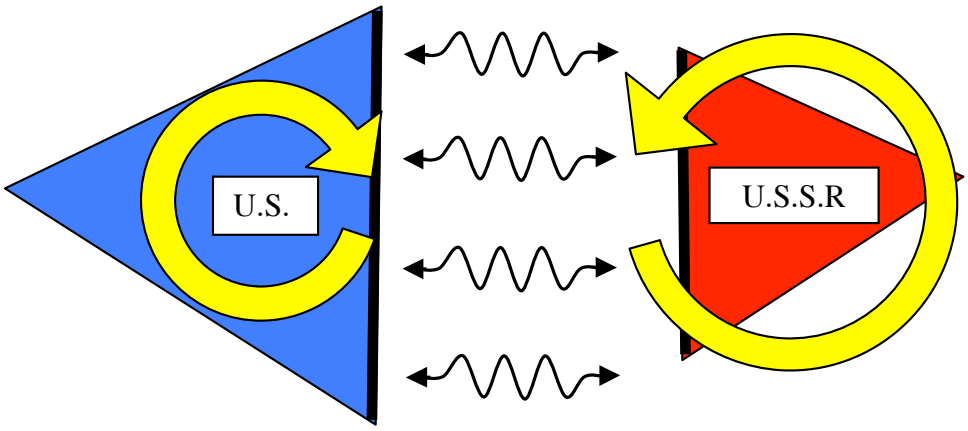


Figure 2.3 Cold War OODA Loops

[4] The United States' capabilities both in terms of intelligence gathering, and in terms of acting on intelligence gathered were somewhat better than those of the Soviet Union. In a competitive sense, the U.S. was bigger (hence the larger triangle). More importantly, the Soviet Union was hampered by a hopelessly slow and ineffective decision cycle caused by excessive bureaucracy, and a pervasive culture of distrust and territoriality throughout the organizational pyramid. Fortunately, the Soviet decision cycle (such as it was) was thus considerably slower than that of the United States. With both capability and adaptability in its favor, the U.S. community gradually settled into a culture of confidence and complacency. The only thing needed was to ensure that the U.S. organization continued to be bigger and more powerful than the Soviet one, since there was no reason for the U.S. to even be aware of, much less consider, decision cycles in its thinking, given its considerable advantages in this area. From these roots came the "big-iron" mentality that now pervades not only the US military, but also the intelligence community. "All we need is more satellites, bigger computer systems, more powerful weapons, more data and we'll stay ahead. We can see them building their silos earlier, building the necessary infrastructure, it will take them years to pose any new threat; we need more of the same only much bigger!" So the thinking went. This thinking may have had some validity in the cold war but it is hopelessly flawed in the new intelligence threat space.

[5] The U.S. intelligence community now faces a multitude of much smaller but rapidly changing threats coming from a highly (often religiously) motivated, diverse set of loosely organized opponents with differing agendas, none of the stability of a state-sponsored organization, and worst of all, with highly devolved command chains and decision cycles that are measured in weeks, if not days, as opposed to the multi-year cycles of the good old days. These lengthy governments cycles have now crystallized as congressional laws and regulations, politics, and the Federal Acquisition Regulations ("FAR"). True, U.S. intelligence capabilities have advanced beyond all measure: the community is now far 'bigger' than before. However, that is of only limited use in this new threat space, and the picture now looks very different:

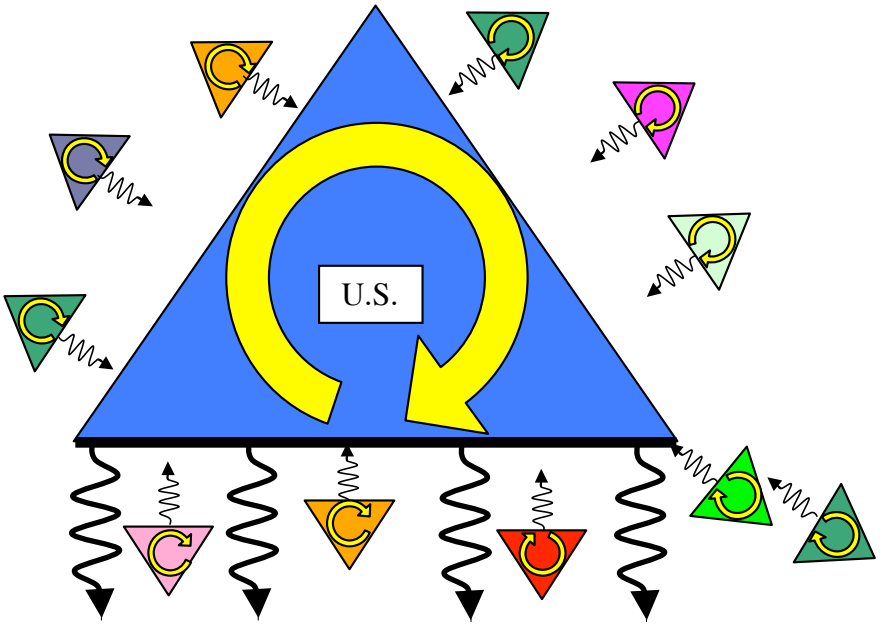


Figure 2.4 Today's OODA Loops

[6] It is like some large animal being attacked by a swarm of bees. The larger combatant cannot track and swat every bee in the swarm because its attention must focus on each in turn, and it takes time to actually swat and kill one. Each individual sting may hurt only a little, and we can certainly swat whatever stings us, but there are plenty more bees where that one came from, and the end result will always be the same. The swarm will eventually kill the animal, no matter how large, and it will finally collapse to the ground surrounded by a mountain of dead and dying bees. All that is needed for the swarm-of-bees strategy to work is a vast supply of worker bees, working largely independently, and eager and willing to die for the cause. Sound familiar? Of course, this is exactly the threat situation we now face. Our nice blue triangle, no matter how large, will eventually look like a piece of Swiss cheese. The swarm-of-bees attack is so effective, not because it brings overwhelming force to bear (the total weight of the attacking bees is still only a tiny fraction of their victim), but simply because it completely overloads the OODA loop of the victim. The problem is the nature of the large combatant. It has a singular focus; that is, it is command or control driven in that the brain must separately and serially issue orders to track and destroy each foe. In the face of hundreds of autonomous foes, such a strategy cannot win. Yet this is exactly the structure we see in our intelligence organizations. How is it possible for intelligence consumers to even be aware of all the potential foes, let alone issue orders to track and destroy them through a command/decision cycle that is measured in weeks or perhaps months? If we consider 9-11 to be one well-placed sting, we can see that we are now facing a swarm of something far deadlier than bees. We are noisily stamping the ground as we do battle with our tiny foes, but we are near to the nest, and our stamping serves only to bring out an endless supply of more and angrier foes. We think perhaps we can track and handle our current problems, but can we really take on the whole hive? We had better be sure, because our noise has surely wakened them!

[7] It is indeed interesting to draw parallels with the reports coming lately from coalition forces attempting to stabilize the situation in Iraq. We see quotes from military commanders stating that they need more localized intelligence. They need systems that can adapt to the situation faster, work more autonomously, and communicate in a more coordinated manner with central authorities. These forces are facing a military threat from exactly the kind of swarm-of-bees opponents that the intelligence community now faces. These are not new lessons learned from new and unprecedented situations. The fact of the matter is these forces are experiencing the full impact of the OODA loop lessons learned and taught by Boyd so long ago. It is to be hoped that the fact that none of these reports is couched in any terms such as OODA loop or decision cycles is because these authors know that their audience may not understand this, and not because our own military has failed to learn from Boyd's teaching. The military is now talking seriously about re-engineering our forces to be more maneuverable, more autonomous, and more adaptive. We should be glad that the military at least has learned (or is re-learning) these lessons. Regrettably the same cannot yet be said for the intelligence community.

[8] What would we ideally like our intelligence cycle and organizations to look like? We can imagine it consisting of a huge triangle comprised hierarchically of many smaller triangles, each of which is attached to the main body, but can swivel independently to orient itself, and is empowered to separately track known or potential foes. Each such triangle must be capable of rapidly passing its findings, without dilution, through the hierarchy to the decision makers and simultaneously to the "swatter" which must be capable of swatting multiple targets at once. In the case of the "swatter", size really does matter, and fortunately for us, the U.S. military, given the political will to use it, constitutes the biggest, baddest swatter on the planet. If we could only track, target, and decide to swat our foes, we certainly have the power to do the swatting.

[9] Why have we not moved to a system like that described above? It is unfair to place all the blame on the intelligence agencies, though they are certainly culpable. After all, these entities can only create such an organizational structure if we give them the considerable technical tools needed to do so. No, on this point, the blame for failure must lie squarely on the technologists and the companies that the government relies on for advice, and to implement such technologies. It is *these* entities, the big primes and the "Beltway Bandits," primarily that share the guilt for our sluggishness.

[10] The key point is that the intelligence cycle itself needs to become a Boyd cycle, and that the speed with which it is possible to iterate through the loop is critical to success. Moreover, we need to realize that this same OODA loop must be practiced at all levels of the intelligence hierarchy if we wish to handle the diverse threat space that exists today. This need for rapid iteration and recursive loop cycling is a key driver for the end-to-end technical approach advocated in this document. By use of such an approach, the barriers between intelligence consumers and those involved in the intelligence process itself can be broken down, and the rapid feedback loop required can be implemented. Most importantly, however, the key lesson of Boyd's teachings is that the ability to rapidly adapt to change is the single most important determinant in any competitive situation. The technology, and the system built on it, must be able to adapt as fast or

faster than the organization that uses it, and certainly as fast as its foes. This key point is either not known, or is completely ignored by the major government systems vendors.

[11] These vendors and systems integrators, oblivious of OODA loop issues, continue to propound the "big-iron" approach that they are so good at, and which as been so kind to them for several decades. The government itself is ill equipped to evaluate or specify technical systems, and so it must either engage these companies and universities to do the job, or it must hire its own capable technical staff to do it. Given the discrepancy between government salaries, and those available in industry, it is not surprising that it is difficult, if not impossible, for the government to hire the huge numbers of technical people that it requires. The government and its contracting community have come up with a neat way around this. The government hires huge numbers of Systems Engineering and Technology Assistance ("SETA") contractors that are in fact employed by the usual vendors, but are then 'leased' to the government for it to use as if they worked directly for it, and thus had the independence necessary to truly examine the technical issues involved. It is these SETA contractors that eventually determine the direction of government technical developments and purchases. It is therefore no surprise that the overwhelming bulk of such contracts finish up getting awarded to the same few players that participate in the SETA pool, and which use the same outmoded (and profitable) big-iron approaches. Moreover, each of these approved vendors have a deliberate policy of hiring senior ex-government people, not only for their experience, which is of course invaluable when dealing with the government, but just as importantly for their contacts. This chain of contacts performs a very effective job of sweeping up virtually all awarded government contracts and ensuring that the usual pool of players remains relatively pure. The intelligence community, with its intense security ethos, and somewhat insular mindset, naturally falls prey to these problems at a differentially higher rate than any other branch of government. Furthermore, the government itself imposes so many hurdles on the participation of small, nimble information companies ranging from its security requirements, to the imposition of its outmoded development models, that it is almost impossible for the intelligence community to take advantage of these small, non-affiliated companies. Yet it is from these smaller, more nimble companies that most of the innovation comes. Because such companies cannot play in the game, they either starve, or more frequently, are swallowed up by the larger vendors in order to acquire their technology. Regrettably, once having eaten a smaller entity, the usual result is simply that the larger entity gets fatter; it seldom acquires any of the nimbleness of its food source.

[12] All these factors combine to maintain the current situation where the government is effectively isolated from the best technical advances in the information realm, and must make do with a sort of distillate of such advances secreted by the usual large vendors. It is not surprising then, that the information systems provided to, or specified by, the government are generally obsolete at the time they are delivered, or shortly thereafter.

2.3 Technology Problems

[1] To answer the question of why our current information systems fail to permit, or even encourage, a more adaptive intelligence infrastructure, we must look at the OODA loop itself in terms of its implications on information systems that support it. Broadly speaking, information systems and information hierarchies can be viewed as a pyramid of knowledge as shown below:

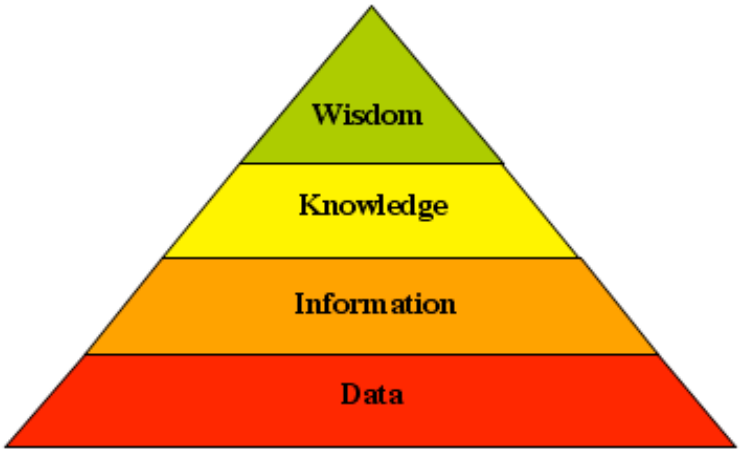


Figure 2.5 Information Systems Knowledge Pyramid

[2] Systems that operate on the **Data** level may be characterized as those that contain or acquire large amounts of measurements or data points concerning the target domain but have not yet organized this data into a human-useable form. **Data**-level systems are most frequently found during the ingestion or data-acquisition phase.

Information-level systems can be characterized as having taken the raw data and placed it into tables or structures that can then be searched, accessed and displayed by the system users. The overwhelming majority of information systems out there today operate in this realm.

A system that operates at the **Knowledge** level has organized the information into richly interrelated forms that are tied directly to a mental model or ontology that expresses the kinds of things that are being discussed in the information, and the kinds of interactions that are occurring between them. An ontology is a formalization of the "mental model" for the types of items that exist in the target domain, and the types of interactions that can occur between them. Few systems today operate at this level, but those that do allow their users to find 'meaning' in the information they contain, and see information and relationships directly in terms of a mental model that relates to real word items of interest.

Finally, we have the level of **Wisdom**. In this domain it is all about patterns within the knowledge. A system operating at the **Wisdom** level allows its users to view new knowledge in terms of their entire repository of past knowledge, to see patterns in that knowledge, and to predict the intent of known or inferred entities of interest that those patterns imply. The key to a wisdom level system is its ability to model what is truly going on, and to predict, by comparison with past patterns, what may be about to happen. Unfortunately, there are no information systems in existence that operate at the **Wisdom** level in any large or generalized domain. As a result, wisdom remains the exclusive purview of the people that use our current information systems.

[3] Let us now examine the OODA loop itself in terms of the types of information systems required to facilitate operation of the loop in the intelligence arena:

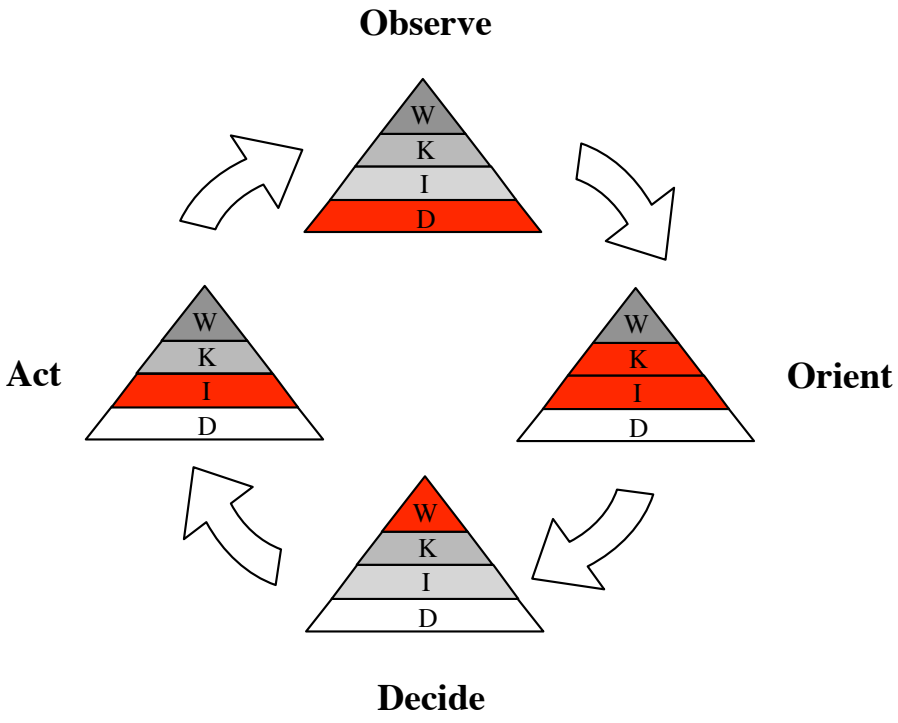


Figure 2.6 Information System Focus at each stage of OODA Loop

[4] **OBSERVE**. It is clear that the process of observation is essentially a **Data**-level undertaking. We must acquire as much knowledge from as many sources as possible as fast as we can. US intelligence agencies, and the deployed systems they use, have certainly got this part of the loop pretty much covered. We are now capable of gathering observations on a global scale, and with a fidelity and diversity that is almost beyond comprehension. We can, and do, also store all this data. There is certainly not a problem here, unless it be running out of storage space.

[5] **ACT**. Once again, we have this portion of the loop pretty effectively covered. To act we must provide easily understood information to those tasked with performing the action. As discussed above, the tools and forces available to the U.S. to act, should it choose to do so, are without equal anywhere in the world. No, we don't have a problem with this part of the loop.

[6] **ORIENT**. When it comes to the process of orienting ourselves to place all our observations in the context of all other pertinent observations in preparation for deciding what to do, we have a serious problem. Orientation not only requires that we convert all the incoming data from every source into a single, unified whole that can be searched consistently by system users (an **Information** level problem), but also because of the huge number of sources and massive diversity of formats they represent. We must convert them into a single common "ontological" form so that the users can compare them one against the other. This requires a **Knowledge**-level system, and yet we know that there are precious few such systems in existence, and certainly none that operates on such a general domain as "understanding what is going on in the world." Our technology has failed to provide for the "Orient" step in the loop. The true state of affairs at this time is that our intelligence organizations have literally thousands of separate mostly archaic **Information**-level systems (or stovepipes), each operating in isolation and focusing on a single source or problem area. Worse yet, the output of these systems is not available to everyone that needs it, and even for those that are available, the interface to, and paradigm for, each system is different, which essentially overloads our analysts by forcing them to learn and serially access a myriad of different systems and to try, often vainly, to integrate all that they have seen in their heads to find meaning. Given the massive amounts of information available, and the tedious interfaces that must be used to access it, our analysts spend the overwhelming majority of their time in the sheer mechanics of accessing the data, and only a tiny fraction in actual analysis (or Orientation). The killing blow at this stage, however, comes from the complete lack of an overarching **Knowledge**-level system to allow analysts to assimilate, compare, and contrast the information coming from all relevant sources expressed in a consistent ontology. Each source presents different information 'chunks,' and any complete integration of the information is difficult, if not impossible. Technology has thus far failed to effectively facilitate the "orient" portion of the intelligence OODA loop. The solution we adopt is to use people, thousands upon thousands of them (since as we have said, each one is so inefficient), to try to close this portion of the loop. The solution, though outrageously expensive, works well enough for tracking small numbers of relatively stable targets, but fails totally (due to the command driven intelligence cycle) when we have large numbers of targets, many of them unknown, that come and go and morph on us continuously.

[7] **DECIDE**. Technology has totally failed to even anticipate, let alone facilitate or assist the decision process of the intelligence OODA loop. The decision process requires **Wisdom**. We must evaluate our new knowledge in light of past experience and patterns, determine what these new patterns mean, and then model the potential consequences of our available actions on the outside world, and on the OODA loop of our enemies. None of these steps is even partially supported by present day systems, let alone supporting all of them in a manner that is tightly linked to the entire OODA loop. When we make decisions to act, we set off a series of events

that themselves represent a secondary OODA loop, they in turn trigger responses that feed back on our own OODA loop and that of others. To model this we must be able to simulate a potential action, insert the, as yet, fictitious information that this action would represent into our repository of the real-world information stream, build models for the likely behaviors and motives of other entities involved in the conflict, and then roll the entire information stream forward to see what might happen. Once we have run this simulation, we must back out all the fictitious information, introduce new information corresponding to other possible choices of action, and then re-enter the cycle as many times as we have possible choices of action. This is exactly what the human mind does when we decide on what actions to take, and it is very good at it. This is the nature of **Wisdom.** But in the intelligence arena, with so many decisions to make, so much information to be evaluated, and such a diversity of bias and opinion, our ability to perform the vital Decide portion of the loop is severely hampered. When one adds in the effects of infrastructure delays, communications problems, misinformation, and rigidity, the situation becomes almost hopeless on this scale. Finding no solution to this problem either technically or organizationally, people have simply learned to live with it.

[8] Over and above the problems we have identified with each individual step in the OODA loop there is a meta-problem that is far worse with our current intelligence apparatus. That is that even if we could grease the inner wheels of each step more effectively, we have no axle to turn our overall OODA wheel. For information and processes to flow around the entire OODA loop rapidly, recursively, and effectively, it is essential that all systems within the loop be able to transparently and pervasively communicate with all others. We need one organization-wide, or more accurately community-wide information system, capable of operating at the **Data**, **Information**, **Knowledg**e, and **Wisdom** levels, designed to provide information to all that need it, in a standardized ontological form, such that all portions of the loop can share and interact with information at any level, examine and validate each others conclusions and the thinking behind them, and introduce new information and knowledge into the cycle. We need a pervasive technology-mediated feedback loop to get the OODA wheel truly started turning. Of course no such framework exists or is even in serious design or development within the community today. Our OODA wheel may as well be bogged down in quicksand, while billions of taxpayer dollars are squandered in feeble efforts, reported as though the necessary progress is being made, to address the threats we see in a post 9/11 World. Not one of these efforts has philosophically faced up to the erosive power of change on itself, let alone on its users.

[9] Thus far we have focused on the technical issues relating to our information system's failure to facilitate the OODA loop of the organizations they are targeted at, but such a focus touches only on a part of the problem. The deeper issue is with our software development methodologies themselves.

[10] As we have stated above, it is obvious that in any system connected to the external world, change is the norm, not the exception. The outside world does not stand still just to make it convenient for us to monitor it. Moreover, in any system involving multiple analysts with divergent requirements, even the data models and requirements of the system itself will be

subject to continuous and pervasive change. By most estimates, more than 90% of the cost and time spent on software is devoted to maintenance and upgrade of the installed system to handle the inevitability of change. Even our most advanced techniques for software design and implementation fail miserably as one scales the system or introduces rapid change. We must realize that in order to facilitate the OODA loop of any organization, the information systems that are employed must themselves be able to iterate through an OODA loop to adapt as fast or faster than the changes that the organization itself is experiencing. Failure to do so will immediately stop the organizational OODA loop from rolling while it waits for its information systems to catch up. In this area, our current technologies fail even more dramatically than they do in addressing the OODA needs of the customer organization. Physician, heal thyself. The reasons for this failure lie in the very nature of the currently accepted software development practice or process. The following graphic illustrates the roots of the problem, which we shall call the "Software Bermuda Triangle" effect.

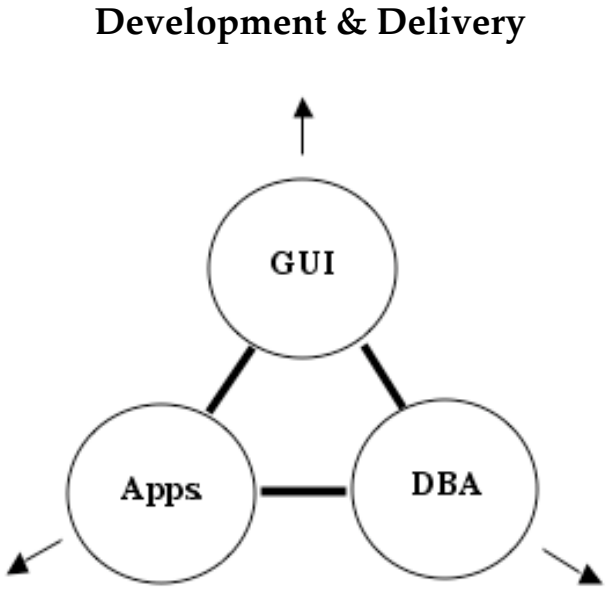


Figure 2.7 The Software Bermuda Triangle - Initial

[11] Conventional programming wisdom (and common sense) holds that during the design phase of an information processing application, programming teams should be split into three basic groups.

> The first group is labeled "DBA" (Database Administrator) in the diagram above. These individuals are experts in database design, optimization, and administration. This group is tasked with defining the database tables, indexes, structures, and querying interfaces based initially on requirements, and later, on requests primarily from the Applications ("Apps") group. These individuals are highly trained in database techniques and tend naturally to pull the design in this direction, as illustrated by the small outward pointing arrow in the first diagram.

The second group is the Graphical User Interface ("GUI") group. The GUI group is tasked with implementing a user interface to the system that operates according the customer's expectations and wishes, and yet complies exactly with the structure of the underlying data (DBA group) and the application behavior (Applications group). The GUI group will have a natural tendency to pull the design in the direction of richer and more elaborate user interfaces.

Finally the Applications group is tasked with implementing the actual functionality required of the system by interfacing with both the DBA and the GUI groups and Applications Programming Interfaces ("API"). This group, like the others, tends to pull things in the direction or more elaborate system specific logic.

Each of these groups tends to have no more than a passing understanding of the issues and needs of the other groups. Thus, during the design phase, and assuming we have strong project and software management that rigidly enforces design procedures, we have a relatively stable triangle where the strong connections enforced between each group by management (represented by the lines joining each group in the first diagram), are able to overcome the outward pull of each member of the triangle. Assuming a stable and unchanging set of requirements, such an arrangement stands a good chance of delivering a system to the customer on time. The reality, however, is that correct operation has been achieved by each of the three groups in the original development team embedding significant amounts of undocumented application, GUI, and database-specific knowledge into all three of the major software components. We now have a ticking bomb comprised of these subtle and largely undocumented relationships just waiting to be triggered. After delivery (the bulk of the software life cycle), in the face of the inevitable changes forced on the system by the passage of time, the system breaks down to yield the situation illustrated below:

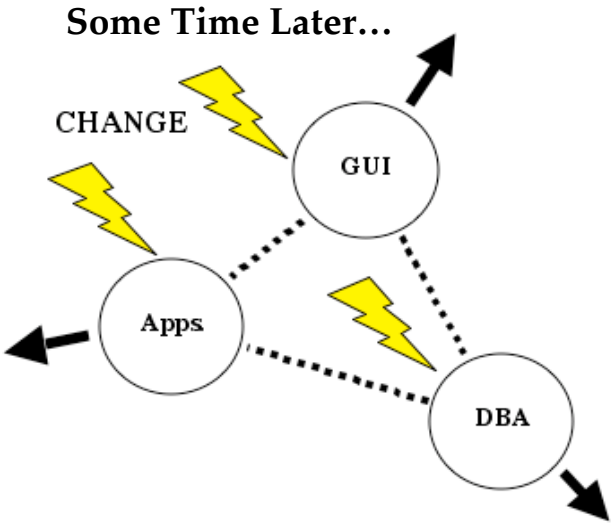


Figure 2.8 The Software Bermuda Triangle – Some Time Later

[12] The state now is that the original team has disbanded and knowledge of the hidden dependencies is gone. Furthermore, management is now in a monitoring mode only (as illustrated by the dotted rather than solid binding forces between the groups). During maintenance and upgrade phases, each change hits primarily one or two of the three groups. Time pressures, and the new development environment, mean that the individual tasked with the change (probably not an original team member) tends to be unaware of the constraints, and naturally pulls outward in his particular direction. The binding forces have now become much weaker and more elastic, while the forces pulling outwards have become much stronger. All it takes is a steady supply of changes impacting this system for it to break apart and tie itself into knots. Some time later, the system grinds to a halt or becomes unworkable or un-modifiable. The customer must either continue to pay progressively more and more outrageous maintenance costs (swamping the original development costs), or must start again from scratch with a new system, and repeat the cycle. The latter approach is often much easier than the former. This effect is central to why software systems are so expensive. Since change of all kinds is pervasive in an intelligence system, an architecture for such systems must find some way to address and eliminate the Software Bermuda Triangle effect.

[13] Over and above the Software Bermuda Triangle effect, another software paradigm related phenomenon contributes to our inability to implement complex unconstrained intelligence systems. In object-oriented programming ("OOP") systems (the current wisdom), key emphasis is placed on the advantages of inheriting behaviors from ancestral classes. This removes the need for derived classes to implement basic methods of the class, allowing them to simply modify the methods as appropriate. This technique yields significant productivity improvements in small to medium sized systems, and is ideally suited to addressing some problem domains, notably the problem of constructing user interfaces. However, as size, complexity, and rate of environmental change are scaled beyond these limits, the OOP technique, rather than helping the situation, serves only to aggravate it. Because the implementation of an object becomes a non-localized phenomenon, tendrils of dependency are created between classes, and the ability of others to rapidly examine a piece of code during the maintenance, and upgrade portion of the development (the bulk of the actual effort) is made more difficult. OOP systems generally introduce the concept of multiple inheritance to handle the fact that most real world objects are not exactly one kind of thing or another, but are rather mixtures of aspects of many classes. Unfortunately, multiple inheritance only makes the scaling problem worse. The maintainer is forced to examine and internalize the operation of all inherited classes before being able to understand the code, and being sure that his change is correct. Worse than this, the 'right' change generally involves changes to the assumptions and implementation of some ancestral class, and this in turn often has a ripple effect on other descendent classes. Eventually, such systems max out at a level of complexity represented roughly by what can fit into a single programmer's brain. While this may be large, it is not large enough to address the complexity of a system for understanding world

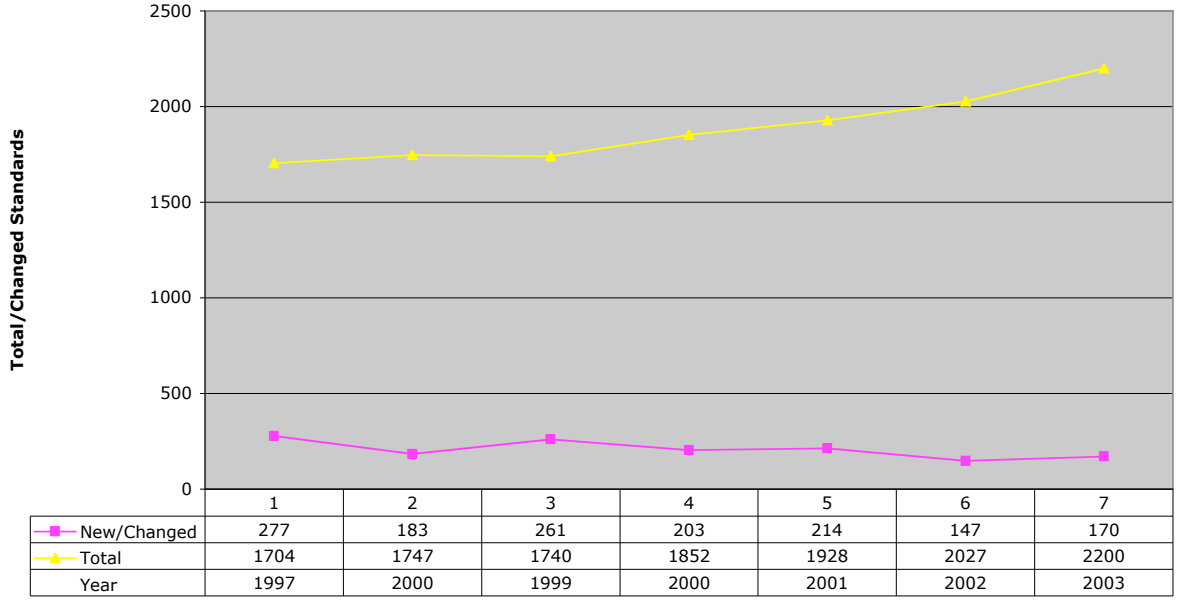
events, and thus an object-oriented approach to attacking such a massive problem is essentially doomed to failure (but only after many years of trying). OOP techniques still rely on the notion of one controlling top-down design. No such design exists in a complex real-world system of this scale. The component-based programming model is slowly replacing OOP as the model of choice in large complex systems. This approach to software specification and development strives to treat software 'modules' much like integrated circuits that can simply be wired together, preferably visually, to create the overall functionality required in a manner more analogous to hardware design (where our efficiencies and process improvements put the advances made in software engineering to shame). This model, though not a complete solution, does at least handle large-scale change more effectively. Since we have said that change is fundamental to the nature of an intelligence system, it is obvious that in addition to all the problems detailed above, we must also move to a totally new software paradigm and methodology if we are to even play in the business of facilitating intelligence OODA loops.

3. Death by Requirements Change – A Case Study

What follows is an in-depth case study. If you want to avoid the gory details, you may wish to skip ahead to Section 4.

[1] To illustrate the problems with designing, implementing, and maintaining a complex multi-source information system in the face of pervasive change, we will study what it might take to create a system capable of monitoring information appearing on the Internet. The Internet is a massively diverse source of timely information on a huge variety of subjects from virtually every perspective. Information on the Internet is characterized by spanning a wide range of qualities, and by containing many contradictory perspectives. Nonetheless, in order to effectively communicate a message in this medium, sources are forced to conform to a fairly limited set of information exchange formats such as HTML, XML, PDF and others. This, combined with on-line accessibility, makes the medium relatively easy to ingest. Internet monitoring is a much simpler problem than those realistically faced by intelligence gathering and analysis systems which must operate over a variety of media and channels, many of which are completely unstructured and not subject to the same standardization processes. As such, choosing to use unified Internet monitoring as an illustrative problem glosses over many of the more complex issues that must be faced by modern intelligence systems. However, as we will see, even in this simplified domain, current approaches fail totally to address the problems of pervasive change.

[2] To model this problem, we first need to get some idea of the rate of change in the "sources" of data appearing on the Internet. Changes in the source of information are of course only part of the changes that any installed system would experience. Another major component, which we shall ignore for this example, is changes in the needs and requirements of the system's users. Obviously, any given web-site can choose to change its layout and content rules at any time, and the true measure of the rate of change in this environment would be infinitely larger than that discussed here. To get a basic idea of the rate of change that must be handled, we will make the simplifying assumption that all change is a result of changes in the information exchange format specifications, and the further simplifying assumption that the changes in the Information Technology ("IT") standards published on an on-going basis by the International Standards Organization ("ISO") constitute the bulk of such changes. Under this assumption, we can examine our proposed system's ability to keep up with just the ISO changes occurring in the IT field. On the ISO website, there is historical information available as to the number of IT related standards published or changed by the organization since 1997. This information is summarized in the graph and data below:



	1	2	3	4	5	6	7
New/Changed	277	183	261	203	214	147	170
Total	1704	1747	1740	1852	1928	2027	2200
Year	1997	2000	1999	2000	2001	2002	2003

Figure 3.1 ISO Information Technology Changes

[3] From the graph (the upper, yellow line), we can see that the total number of standards governing the environment is increasing continuously. Moreover, between 2001 and 2003 more new standards were added than in the entire time since records were made available; that is, the rate of change is increasing dramatically. However, the more significant realization comes when we look at the number of changes occurring per year (the lower, purple line). From this we see that the average rate of change over the period is roughly 200 changes per year, or one new or changed standard every 1.7 days! This is a truly massive rate of change, and it is this figure (which our current design approaches largely ignore), that leads to the most significant problems downstream.

3.1 The Classical Approach

[1] To evaluate the impact of this change on a system designed to monitor evolving Internet content, we must next look at the typical productivity rates achieved by programmers under current development approaches. Fortunately, there is very thorough data available on programmer productivity derived from a number of studies on the subject. For more details, the reader can refer to the extensive literature. However, for the purposes of this comparison, the author has used figures given in the book "**A Manager's Guide to Software Engineering"** by Roger S. Pressman (McGraw Hill 1993), which represents a fairly complete study of the field. On page 59 of this book, there is a table giving the typical range of programmer productivity (measured in Lines Of Code – "LOC") as a function of the type of application being developed:

Application	LOC/person month
Information Systems	800-3200
System Applications	400-1000
Real-Time Embedded	100-600
Human-rated Systems	30-400

Table 3.1 Typical LOC/person month by Application Type

[2] Table 3.1 above illustrates that the expected productivity of a programmer varies widely depending on the nature of the application involved with the highest levels occurring in relatively simple systems (such as database front-ends), and the lowest occurring in embedded and human-rated systems. The nature of the problem faced by an intelligence system places it firmly in the "System Applications" category since the necessity to integrate and rationalize information across a variety of external sources removes the huge simplification implied by the "Information Systems" category, and moves the problem more towards the realm of responding to outside change coming from diverse sources, which is essentially what characterizes the "Embedded" domain. Individual programmers, working within frameworks of their own devising, can often achieve productivity rates far higher than those given. However, we must discount these figures based simply on the sheer size of the system required, and the obvious need for a large team to address such a ferocious rate of change.

[3] As application size gets larger, the complexity of making a change to the existing code base causes productivity to drop off rapidly from the ideal above. On page 60 of the same book, we find the following diagram illustrating the relationship between programmer productivity, measured in Function-Points ("FP") per person-month, and total application size (measured in FP).

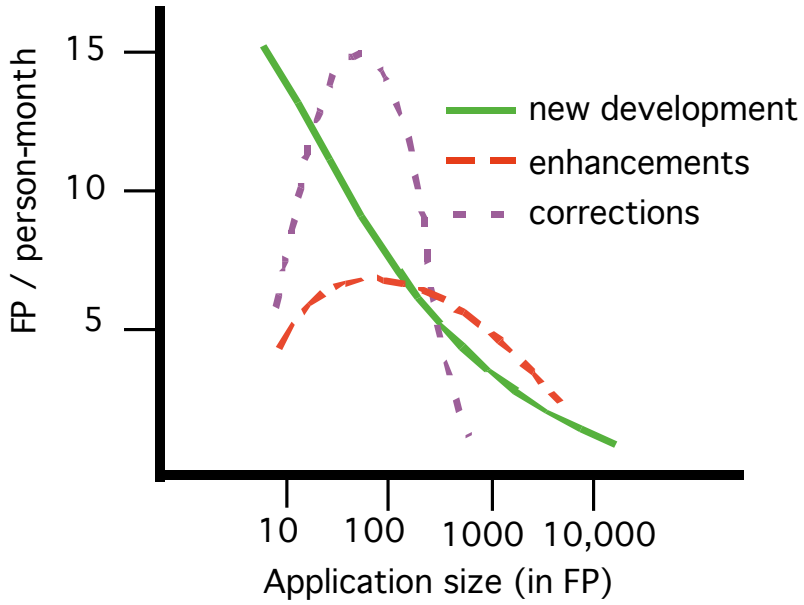


Figure 3.2 Classic Software Development Productivity Curves

[4] The diagram above illustrates a number of significant points, and is essentially at the root of why current design strategies focus on a requirements-based waterfall driven approach to software development. As can be seen (green line), programmer productivity is at its highest when developing new code in response to a stable set of requirements and absent any need to integrate with existing constraints. Under this scenario, a programmer productivity of at least two to three times that found in other phases can be realized. For this reason, current software development strategies, quite rightly, focus on placing most of the work in the development of an initial set of well defined and un-ambiguous requirements, and the successive refinement of those requirements through design, and finally to implemented code. Given a stable requirement set, the program manager can expect an overall productivity on the green (new development) curve above, and hopefully can bring the project to completion, and installation on time and on budget. The rapid tail-off shown in the purple (corrections) curve illustrates what happens to programmer productivity if the requirements are changed during development, and more importantly, also applies throughout the maintenance phase of the software life-cycle. This is caused by the need for the programmer making the change to understand the previous code and how it fits into the complete system prior to making any change, and the fact that even with the best of intent, it is difficult to avoid tripping up over some undocumented code dependency when changing code within a large system, especially if the original code is poorly documented or the original programmer's style differs significantly from that of the one making the change (almost always the case given the nature of programmers who invariably prefer to re-write rather than try to understand). The initial spiked shape of the corrections and enhancements curves at first seem counter-intuitive until one realizes that, for very simple programs (up to say 100 FP), it is often the case that the original implementer failed to put in place the right abstraction layers (or even commentary) since it did not seem worth the trouble. This means that these simple programs can be even worse hit by

requirements change (which tend to impact in multiple places now due to the lack of abstraction) than larger programs.

The differences between these curves almost entirely accounts for the well-known fact that 90% of all the costs in the life cycle of a piece of software occur during the maintenance phase. It is during the maintenance phase that the requirements change the most, and since responding to these changes occurs entirely on the "corrections" line, the cost of such modifications can be astronomical compared even to the original development costs. This phenomenon accounts for why most large software systems rapidly become obsolete and fail to adapt to changing needs, thus invariably forcing the creation of a "new improved" system from scratch that holds the promise of being developed on the "green" line, and of incorporating all known requirements out-of-the-box. Of course the truth of the matter is that as total system size increases, and in the presence of continuing requirement change, the time taken to actually develop and integrate the "new improved" system, means that by the time it is installed, it already does not meet the changed requirements, and we get into the classic cycle we see in all large software system developments of being over budget, late, and eventually themselves becoming a target for the "if only we could start over again" cycle. Down this sink-hole countless billions of government (i.e., taxpayer) dollars have been poured and continue to be poured today.

[5] The function-point measure (FP) is a standard means of measuring the complexity of software that is to a large extent independent of the LOC measure. The reason software engineering has moved from LOC-based productivity measures to FP-based is due to the massive variations in the number of lines of code it takes to implement a given functional requirement between different programming languages and methodologies. To illustrate this, the following table (from the same source) gives the average number of LOC it requires to implement one FP as a function of the language used:

Language	LOC/FP (average)
Assembly Language	300
C	120
COBOL (obsolete)	100
FORTRAN (obsolete)	100
Pascal (obsolete)	90
Ada	70
Object-oriented languages	30
Fourth-generation languages (4GL)	20
Code generators	15

Table 3.2 Lines of Code/Function Points for Various Programming Languages

[6] Table 3.2 above leads to a very simple formula for estimating the complexity of a given application in FP given knowledge of the language used and the number of lines of code as follows:

$$\mathbf{FP = LOC_{appl} / [(LOC/FP) * BAF]}$$

Where LOC_{appl} is the total number of lines of code in the application and LOC/FP is the appropriate value from the table above given the implementation language. The Backfiring Adjustment Factor ("BAF") is a function of the complexity of the application involved as follows:

Complexity	BAF
Very Simple	0.7
Simple	0.85
Average	1
Moderately Complex	1.2
Complex	1.3

Table 3.3 Backfire Adjustment Factors for Differing Systems Complexity

Obviously, in a large intelligence system we should use 1.3 as the value for BAF.

[7] We can use all the information above combined with actual LOC counts from the existing Mitopia® architecture to come up with a minimum FP measure for implementing a large intelligence system. Mitopia® has approximately 2 million lines of code (written entirely in C) at this time. It makes use of no external technologies (other than the underlying OS), including implementing its own integrated distributed and federated database model, so we can estimate total FP complexity without the need to introduce (debatable) fudge-factors to account for these external technologies. Given this, we arrive at an FP measure for Mitopia® of 12,800 FP. As can be seen, this is an extremely large and complex system, since the programmer FP productivity graph given above contains no data for anything with an FP measure above 10,000, and would imply that FP productivities at this scale have essentially fallen to zero. Furthermore, Mitopia® is the result of over ten years of evolution towards a fully architectural (rather than application-specific) approach, and thus contains significantly less lines of code than it did in the past (perhaps three times as many lines at its peak). Because of the layered architectural nature of Mitopia®, actual FP numbers are almost certainly far higher than those given in table 3.2 above. For simplicity, we will ignore this fact. New system developments, given the constraints of delivery schedules, do not have the luxury of developing and refining an architectural (largely codeless) approach to application development, and thus are likely to require at least 5 million lines of (C-equivalent) code to attain similar functionality. This leads to a more realistic estimate of the FP for developing such a system from scratch of around 30,000. This is so far off the productivity scale that given just this information we can confidently say that implementation and installation of such a system from scratch is likely to be a very expensive and lengthy process. To avoid charges of bias in the analysis, we will use the figure of 12,800 for our analysis,

despite the fact that this is highly unrealistic in the presence of compressed delivery schedules. Given this, if we assume an optimistic productivity of 10 FP/person-month (which is only valid for trivial systems), we still come up with the fact that we need 1,280 person-months to implement the system. If we assume a programming team of 50 people, with no delays, hitches, or requirements changes, we could hope to implement the first prototype of such a system in a little over two years.

[8] What we have estimated here represents the cost and time taken to initially implement the equivalent of the Mitopia® architecture. Mitopia® itself does not implement any ISO standards. It represents only the task to implement just the basic housekeeping software necessary to form the substrate on which such a standards development task could realistically be expected to succeed. We must now add to the initial costs the time taken to actually implement the ISO standards themselves, regardless of the underlying system architecture. To estimate the impact of this, let us assume that to implement a system compliant with any given ISO standard among those we are tracking requires roughly 10,000 LOC/standard. We can then multiply this number by the roughly 2,200 existing ISO standards. This yields a total development effort for the standards or application specific team (as distinct from the 50 man underlying architecture team) of 22 million LOC. This is considerably larger in magnitude (but simpler) than the task facing the architecture group (around 2 million LOC), so we can assume that we will need a second, larger standards development team to complete the task. We now have a total FP number for standards development of:

$$\mathbf{FP}_{\text{standards}} = (2{,}200 * 10{,}000) / (120 * 1.3) = 140{,}000$$

At a 10 FP/person-month rate (perhaps more realistic in this case), we will need 14,100 person-months to implement the standards. However, in the standards development case, the complexity of the programming task more closely approximates that of an "average" complexity development, since each standard development can be expected to be relatively independent of all the other standards (assuming a good underlying substrate). Let us, therefore, divide this estimate by a factor of 10 based on the relative simplicity, and on the assumption that there will be cross-fertilization between the various standards implementers so efficiency in this case may be significantly higher. We are still left with 1,410 person-months of development. If we want to also get this done in the same 2-year period, we see that we will need a standards programming team of around 60 people.

[9] We have estimated the programming effort required to implement an internet monitoring system above, but this calculation as explained before is based on maximally productive teams totaling 110 (50+60) people, and we know that such productivity can only be achieved in the presence of a well developed and stable initial set of requirements. We must now estimate the time and effort necessary to come up with such a set of requirements, and for the project to succeed; this time must be spent before any programming effort begins. Furthermore, we have thus far ignored the time taken to actually integrate and test the software into a system. Since we know the programming effort, we can use the extensive data collected in the literature to

estimate the efforts required for these other phases. In the book "**Software Engineering**" by I. Sommerville (International Computer Science Series 1985) we find the following table, which occurs, in a similar form in most other software engineering references:

System Type	Phase costs (%)		
	Reqs./Design	Implementation	Testing
Command/Control Systems	46	20	34
Spaceborne Systems	34	20	46
Operating Systems	33	17	50
Scientific Systems	44	26	30
Business Systems	44	28	28

Table 3.4 Software Effort by Project Phase

Table 3.4 above indicates that we should assume that the programming effort constitutes roughly 25% of the total implementation time/costs; the requirements phase requires around 40% and the testing/integration phase the remaining 35%. If we assume that these tasks fall entirely on the 50 man "architecture" team (assuming implementing standards becomes a relatively rote and reliable process), then if we spend 2 years on programming, we must first spend around 3 years on system analysis, requirements development and initial design. After the programming is complete, a further 3 years of testing will be required before the prototype system can be considered functional. We are now looking at a total elapsed time to develop and install such a system of around 8 years, which matches closely what we would expect, given the history of past large-scale projects. But we have now extended our time-line from two to eight years, and so given the continuing rate of requirements changes, we are exposed to four times the amount of change during the development phase.

[10] Continuing with our ISO changes scenario for a moment, we know that while these teams of 50 and 60 people are slaving away for eight years to develop the initial prototype, roughly 16,000 new or modified ISO specs will have been released. This of course will mean additional delay to the schedule as the new requirements are folded into the now implemented software. Let us further assume that in reality only 10% of any given standard changes with any update. Given the change rate of roughly 17 standards changes per month, we can derive the FP impact per month from the changes using our formula as follows:

FP impact/month = (17 * 10,000 * 0.1) / (120 * 1.3) = 108 FP/month

But this FP impact is on the "changes/corrections" graph (fig. 3.2) so we must estimate a much lower FP rate per programmer to keep up with this change. Let us generously estimate an FP rate for such changes of 2/person-month even though the graph tells us that such productivity will be almost impossible to achieve. This indicates that simply to keep up with the ongoing rate of change in the requirements, we will need a programming staff of around 50 people. This

is, of course, in addition to the original staff of 110 necessary to implement the initial system in the eight-year timeframe. Our simplistic programming manpower-loading curve now looks as follows:

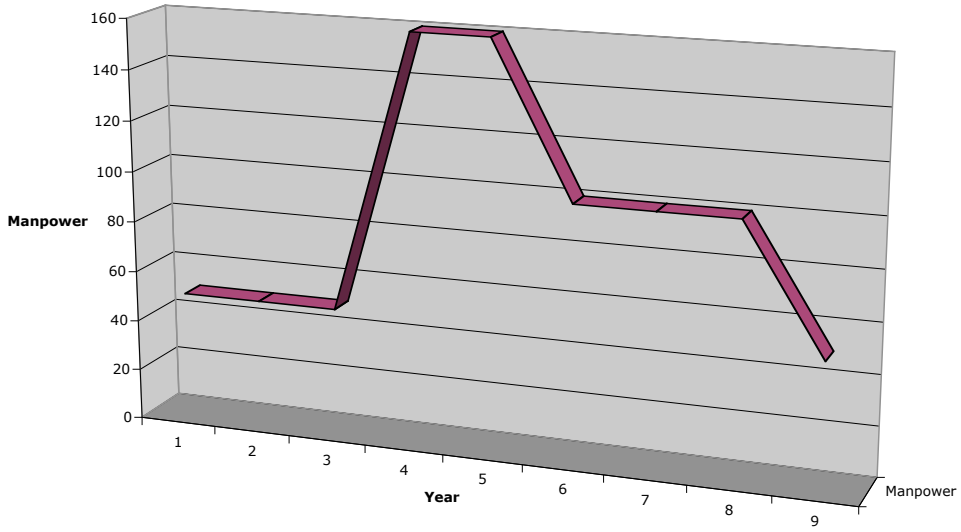


Figure 3.3 Classical Manpower-loading Curve

Assuming that it were in fact possible to coordinate three teams of 160 total people working on the same integrated code base (which of course it is not), we can now come up with a rough estimate of the cost to the eventual customer, just in programming time, of such an undertaking. We will take a burdened cost to the customer of $120/man-hour (160 hours per man month) as the basis for our estimate, although the actual cost would likely be far higher given the unique nature of the people necessary to succeed in such an intense environment. This gives us a minimum programming-only cost for initial implementation of:

(160 persons *120 *160 *24) + (50 persons * 120 * 160 * 36) + (100 persons * 120 * 160 * 24)

The total is $160 million over 8 years, with an on-going maintenance programming cost of $25 million per year for each year after initial system delivery. Given the unrealistic nature of the assumptions that lead up to this estimate, the actual figures would likely be many times that, perhaps even in the billions, and the delivery schedule far longer (which of course makes the situation worse as far as the impact of changes). The risk of failure in this enterprise is extremely high. Our simplifying assumptions and optimistic formulae do not reflect the actuality of real software development. The chances of actually delivering such a system using this approach on schedule, and to budget, are slim to none.

[11] Other than the catastrophic assumption that standards development can proceed in parallel with architecture development, perhaps our worst assumption above is the fact that it is actually possible to achieve productivity out of such large development teams. Let us examine the

dynamics of the 50-person maintenance team more closely, given industry experience of the impact on programmer productivity of the communications overhead implied by the size of the programming team in which they operate. Remember that, since this project is targeting a diverse set of standards striking all aspects of the system (architecture and standards) equally on a continuous basis, the impact of which must be integrated into a single unified whole in the control-flow based code base, we are essentially forced to have such a large team, and breaking the team up into smaller units will serve only to decrease productivity by raising additional communications and infrastructure barriers. It has been shown that the impact on a given programmer's productivity of say 5,000 LOC/year is reduced by roughly 250 LOC per communications link between that programmer and the other members of the team (see page 276 of the referenced book). This simply reflects the time taken to meet, discuss, agree and argue about implementation approaches, and is an unavoidable penalty of increasing programming team size. For a programming team with 'N' members, the number of communications paths is given by SUM(1..N) of (N-1), so we finish up with the results showing in Table 3.5 below:

Programmers	LOC/programmer	# links
1	5000	0
2	4875	1
3	4750	3
4	4625	6
5	4500	10
8	4125	28
10	3900	44
20	2687	185

Table 3.5 Productivity Degradation with Team Size

If we extrapolate this series, we see that in fact, productivity per-programmer, when one takes into account the impact of operating in a large team, effectively drops to zero once the team size reaches around 45 people. In reality the drop-off is much faster, and few people have succeeded in maintaining any serious productivity with integrated teams any larger than 25 persons. It is a pre-requisite of the target problem domain that it can be effectively partitioned into multiple distinct 'technologies' having very little overlap in order to bring more programming effort to effectively bear on the problem. Unfortunately, our integrated intelligence environment, under a continuous externally driven change of requirements, cannot tolerate such a criteria, we must have a maximally productive team of 50 people to keep pace.

We are thus driven to conclude that, given conventional approaches to developing such a system, even if a system to "monitor internet content" could be initially specified and developed, it is essentially mathematically impossible to succeed in maintaining such a system to keep pace with the changing environment. Once again, such a conclusion comes by reducing the problem to tracking changing ISO standards, which is of course a trivial subset of the true problem being

faced by such a system. In essence, if we cannot change the ground rules for how we develop intelligence systems on this scale, we are doomed to fail in their implementation, no matter how much money we throw at the problem. This is a very discouraging prospect given how critical it is that we find some way to succeed in this endeavor.

3.2 The Mitopia-based Approach

[1] The differences in the technical approach taken by the Mitopia® architecture have been briefly touched-on in the discussions above, and a full detailing of all the issues and the strategies MitoSystems has adopted to address them is beyond the scope of this document (though available). Put in its most concise form, Mitopia® is an architecture based largely on the premise that an ontology, and the data expressed in that ontology, not the program, should be in control of specifying and extending system capabilities, persistent storage, and user interface. This eliminates the "Bermuda Triangle" problem that is at the root of the punitive FP productivity graphs we see for standard software "changes" approaches (as illustrated in the earlier graph). As such, the Mitopia® environment is split into two main portions. The Architectural Software (implemented and updated by MitoSystems), and the Configuration Software which can be modified and extended by other developers and/or the target customer with comparatively little training and technical expertise. Due largely to the incorporation of an ontology-definition language (C*), and the provision of an extremely powerful data/ontology driven front-end ingestion architecture known as MitoMine™, the overwhelming majority of changes to the system in response to new requirements require no compiled code changes to implement whatsoever, merely adjustment of the front-end scripts and/or the ontology. In MitoSystems current installations, the system specific compiled code that customizes the architecture to the specific application typically requires around 100KB out of a total application size of 54MB. That is less than 0.2% of the code required to implement the system. Given this fact, we should expect an increase in programmer/maintainer productivity of a factor of roughly 500 over conventional approaches when it comes to adapting the system to on-going change. To validate that these kinds of numbers are actually achieved, we can examine actual implementation times and configuration script LOC counts from past developments in response to new source format specifications.

[2] MitoSystems currently delivers 40 different fully implemented MitoMine™ scripts as part of its initial example set to allow developers to acquire new sources and to illustrate many of the techniques that can be applied within MitoMine™ to do this. Each script in this set represents all that is necessary to ingest the equivalent of one ISO IT standard and to convert it into the system ontology, thereby automatically implementing, without additional compiled code, all required system storage, searching, user-interface and other functionality via the technologies within Mitopia®. In fact, many of these scripts relate to the ingestion of highly un-structured sources such as publications, and this is considerably more difficult than a well-structured ISO standard designed from the outset for computerized processing. Many scripts also relate to ingesting sources in foreign languages, particularly Arabic, this is relatively trivial in MitoMine™ but would of course have a tremendous negative impact on a conventional approach. Some even relate to ingestion directly from encoded binary sources, again supported in MitoMine™, but a very different matter with a conventional approach. Examining historical development times for these scripts, we see that each such script takes an average of four days to develop, test, and install into the target system. Some scripts take only hours, others many weeks. This is an end-

to-end figure, and thus represents an accurate measure of the ability to keep up with on-going change. The total LOC count for all these scripts taken together is 3,495 (.BNF) plus 360 (.LEX onecat) for a total of 3,855 LOC, that is an average of 96 LOC/standard. Contrast this to the equivalent figure to implement the same functionality in a conventional system (at 10K LOC/standard) that is 400,000 LOC. This is a LOC improvement by a factor of over 100. The total time to implement these lines gives us a Mitopia® programmer productivity when configuring the system of 771 LOC per person-month. This is roughly the same as the LOC values for other approaches. However, it is when we look at the more important FP numbers that we see a truly startling difference.

[3] From the discussion above, we know that the total FP to implement these standards in a conventional approach is given by:

FP = (40 * 10,000) / (120 * 1.3) = 2,560

Thus, we can calculate the value for LOC/FP in this case for the Mitopia® environment as 3,855/2,560 LOC/FP that is 1.5 LOC/FP! The best/lowest LOC/FP number we can achieve with any other known approach is 15 for code generators, but these are restricted to relatively trivial domains, and thus cannot realistically be applied in this case. The nearest practical number we could use in such an application is that for object-oriented languages, which is 30. Thus, we are seeing an increase in LOC/FP productivity of at least a factor of 20, and more likely closer to 80 over a conventional approach.

[4] Just as we did for the conventional approach, we are now in a position to calculate the size of the programmer team necessary to initially implement, and to subsequently maintain, the "internet monitoring" application of our example using the Mitopia®-based approach.

[5] To implement a system to handle the current set of some 2,200 ISO IT standards, from scratch (and considering only programmer time), based on the Mitopia® architecture, we will require (2,200 * 96) LOC. Using a figure of 771 LOC/person-month, this can be expected to take us 274 person-months. If we want to get this done in two years, we will need a team of just 12 programmers. This is a much more practical proposition than the 110 person team we required before. More importantly, as for the conventional approach, we must include the additional manpower required to keep up with the 105 FP/month change rate during that two-year period. At a rate of 771 LOC/person-month and 1.5 LOC/FP, this indicates that we will need just one additional programmer to easily keep pace with the changes. Given this staffing, we can use the same hourly rates used in the conventional approach to come up with an initial system development cost (programming only) of around 120 * 13 * 160 * 24, or roughly $6 million dollars (plus, of course, the cost of Mitopia® itself). Thereafter, our maintenance cost (assuming, for safety's sake, we have two full time maintenance programmers) would be roughly $500,000 per year (plus the cost of a continuing Mitopia® maintenance agreement/license). Manpower loading would appear as follows:

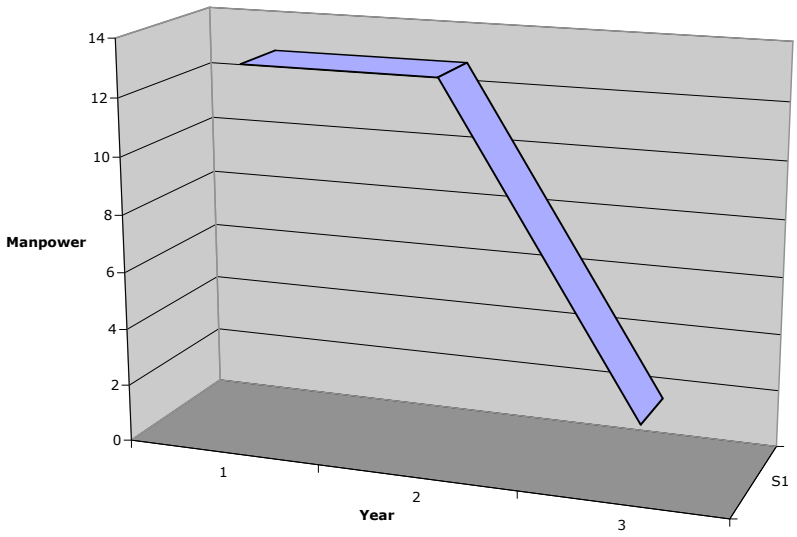


Figure 3.4 Mitopia® Manpower-loading Curve

[6] Should we degrade these figures based on the size of the programming teams as we did in the first example? The answer is probably not. For the maintenance team of two people, this impact is obviously negligible, and for the 12-man development team the situation is now completely different because, as in the standard development process example, each member of the team developing the ingestion scripts for the ISO standards is not actually writing code, and each script is totally independent of every other script, thus we do not expect to see the conventional slowdown (except perhaps as time wasted at the water cooler). Note that compared to the 50 man team needed to keep pace with ongoing change in the conventional approach, we now need 1/5 of a man, that is an increase in productivity of around 250 times (with a corresponding reduction in maintenance cost). This is not far off the ideal improvement we expected from our simple estimates above of a factor of 500. More importantly, if we tie this to the improvements to system utility to the end customer through a corresponding increase in the cycle rate for OODA loop of the system itself, the benefits of a Mitopia®-like approach become truly apparent. The OODA loop improvements with a Mitopia® approach come mainly from the fact that customer staff themselves, once trained, are capable of maintaining and updating the system. There is no need to go back to the original vendor with costly and lengthy change-orders to get things done, as there is in a conventional system. In a change-order based scenario, the minimum realistic time to implement system "upgrades" is on the order of 3 to 6 months. With the Mitopia® - based approach, the turn-around time to incorporate a change when made by the customer staff themselves is measured in days.

[7] But we have thus far ignored the requirements development and testing phase. Why? The answer for the testing phase is simple: in our first example we also ignored the testing phase for the standards development, because the assumption was that standards implementation is relatively simple given a stable substrate. If we approach the problem using the Mitopia® architecture as our substrate, our programming effort consists only of standards development.

MitoSystems has already been through the requirements, development, and testing phases for the architectural portion, and thus we can ignore testing or requirements just as we did in the first example. The same applies to the requirements development phase. MitoSystems has already done that, so we can ignore it. This seems somehow counter-intuitive. What we are effectively proposing here is a software life cycle consisting only of coding and maintenance. How can this possibly be? Surely there must be at least some requirements phase prior to standards implementation. The answer to this question lies in re-examining the FP/person-month graph presented earlier, but now including the FP numbers for a Mitopia-based approach. We have seen that Mitopia® has a 771 LOC/person-month figure with a 1.5 LOC/FP factor (for an FP/person-month of around 500) in this domain. But these figures are independent of the new-development/corrections issue. After all, all that is required to keep track of changes is the modification of existing, very small, and completely isolated MitoMine™ scripts. We, therefore, expect little or no difference between initial development and maintenance, and no significant fall-off in productivity with total system size (as measured by the number of standards addressed). Our modified (and re-scaled) FP productivity graph now appears as follows:

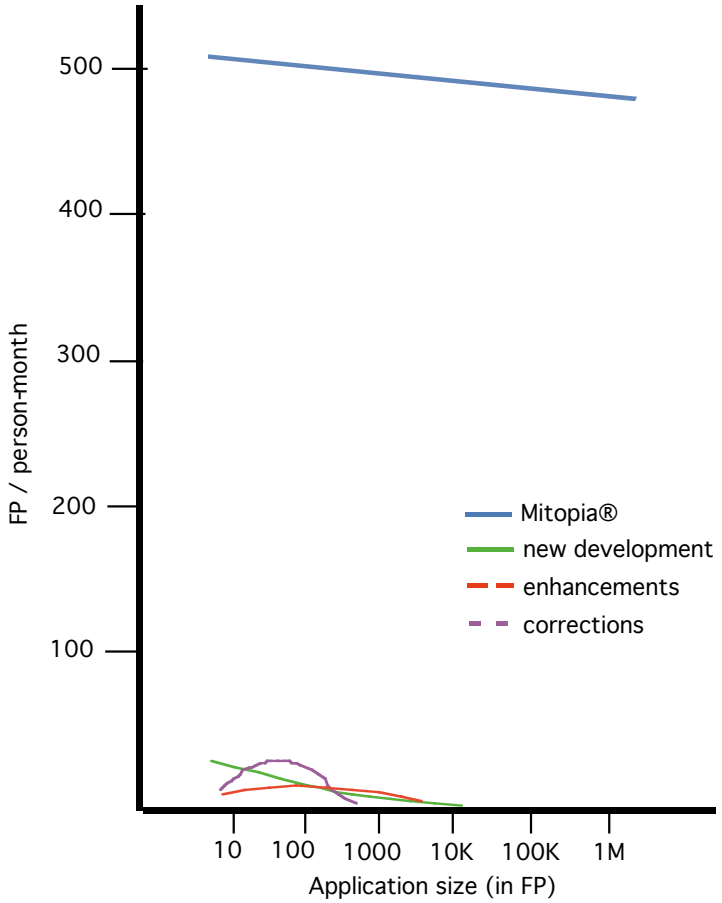


Figure 3.5 Updated Software Development Productivity Curves

[8] As we can see, the revised scale of the graph, and the fact that the Mitopia® curve, in this domain, is independent of project phase, makes nonsense of the conventional wisdom that we must first sit down and develop a stable set of requirements before we proceed to code. We can see that we pay essentially no penalty for diving immediately into coding without any knowledge of the full scale of the problem, it simply makes no difference. At least in the domain studied, it is clear that we are now operating in new and uncharted territory and that we have cast aside most, if not all, the constraints that previously held us back.

Metric	Current Approach	Mitopia® Approach	Ratio
Schedule	8 Years	2 years	**25**%
Manpower	160 People	13 People	**8**%
Cost	$160 Million	$6 Million	**4**%
Maintenance	$25 Million Annually	$500,000 Annually	**2**%

Table 3.6 Comparison of Cost, Manpower and Time for Mitopia® vs. Current Approaches

[9] Are the Mitopia® estimates in Table 3.6 above optimistic? Probably. But they are certainly far less optimistic and unrealistic than the assumptions we had to swallow with the classical development example, so the relative conclusions we have drawn still stand.

3.3 Other Approaches?

[1] As in any field, for every problem there is rarely a shortage of people, companies, or groups that claim to hold the key to some or all of the solution. So it is with our information system problems. Since 9-11, it must indeed be a noisy world in the field of intelligence systems acquisition. Every person or company, no matter how poorly qualified, is out there re-packaging or just proposing something they claim (without proof) will help solve the problem. They all pursue the grail of those promised billions of new intelligence and homeland defense spending. The usual players are in there too. They may for the most part be poorly qualified in this field, but they certainly want to make sure their government food source does not get nibbled at by others. The noise must be deafening! How to sift through this over-abundance of proffered help for the few nuggets that are undoubtedly hidden within it. The swarm of bees is back, this time unwittingly attacking and overwhelming the OODA loop of the very group they are trying to help. The result is that government IT policy at the moment behaves much like a family of rabbits caught in the middle of the road by oncoming headlights. Each family member moves a few small steps in one direction, looses its nerve, and then starts off in another direction, bumping into each other as they go. Meanwhile the headlights grow ever larger! Many more billions spent for little or no progress.

[2] However, we are focused in this paper purely on systems or technical approaches that claim to solve the architecture-level problems that plague us. At this level, things are much quieter and there are relatively few approaches that have even the slightest credibility in this game. The author has spent much time (since long before 9-11) with technical people from a variety of intelligence agencies discussing exactly these kinds of problems, and as such is probably as well equipped as anyone to provide a survey of the approaches that are being considered within the community. They are as follows (in no particular order):

(a) Same as usual approaches (but bigger) based directly on relational databases, or perhaps using object-oriented languages and/or object-model databases to spice things up. Frequently these approaches advocate cobbling together a variety of Commercial Off-The-Shelf ("COTS") applications to accomplish their goals.

(b) Corporate Information Factory ("CIF") based approaches.

(c) Approaches using the Associative Data Model.

(d) Mitopia®. This approach has been detailed elsewhere in this paper and will not be discussed further here.

[3] Those that propound the first option, that is business as usual only bigger, are simply showing us that they are either ignorant of, or don't care about, the true nature of the problem. They will inevitably fail. We have explored some of the reasons behind this inevitable failure in excruciating depth in our case study above. It is almost not worth commenting on such

strategies for fear of inadvertently giving them credibility. The only reason to do so is that regrettably, this strategy seems to be the overwhelming favorite still. The number of new information initiatives within the intelligence community that have simply lumbered into motion in this direction is truly startling. These initiatives are in progress as we speak and together they will account for billions of dollars of wasted spending. Over and above the developmental problems these systems will face, the choice of a relational database as the metaphor for persistent storage is a horrible mistake: I know this through personal experience. For many years, Mitopia® was layered on top of a relational database (in this case Oracle), not because of any belief in the merits of the relational model, but simply out of pragmatic acceptance of the fact that customers are comfortable with and understand relational databases. The relational model is incapable of rising even to the Knowledge level, let alone Wisdom. To achieve this, one must build programming edifices that turn an archaic model based on the assumption that everything in the world can be represented as a set of matrices with fixed sized cells, into something that can more closely represent reality and the complexity and diversity of the relationships it contains. This code edifice must constantly do battle with the weakness of its underpinnings. This discordance, and the mountains of application-specific (and inflexible) code that must be written to cover it up, can spell the death knell of any project that attempts it, especially when requirements change. Once again, this is not theory. This advice comes from bitter experience treading this road for nearly ten years. Just say no!

[4] A few groups are clearly aware that the relational model, which is after all more than forty years old, is dead. Instead, they propose "business as usual," but let us replace the relational model with the object model. We still see companies claiming that the object model (or at least their implementation thereof) is the "new thing," even though the object model itself is now celebrating its quarter-century. The object model is an attempt to graft the benefits of object-oriented programming (which, by the way, first appeared in the late 60's, almost co-temporal with the relational model), into the database domain. The idea is that by having a database model that is object based, it is easier to tie this to code that is object oriented and, thereby one should realize considerable productivity benefits. By around 1990, object-orientation had become "flavor of the month," and this in turn led to a relative explosion in object-model database products. Since then there have been some successes, but the object model has basically failed to make even the slightest dent in the relational/SQL armor. The reasons are many, but the greatest weakness of the object model reflects one of the great weaknesses of the object-oriented paradigm, and that is that as with OOP, the object model surrounds all data items with an access process, much like an OOP class. All access to and from the internal content of the data must pass through this guardian. The benefits, as for OOP, are that with such controlled access, it is far more difficult for the unwary to mess things up. The downside, and it is a fatal one (both for the object model and for OOP), is that when you scale things up to the truly massive, the need to go through this guardian to get anything done means that code that must do the kinds of global operations implied by database access or complex intelligence analysis, must do so through the guardians, and thus cannot be as highly optimized as it needs to be with scale. The result is the same for object-model databases and for object-oriented languages; eventually they run out of speed and slowly (and of course very safely) grind to a

halt. Java is the ultimate expression of this moribund philosophy. You must go through an interpreter, and you cannot even directly access memory! Very clever. Very safe. Verrrry slow! Very silly. Not surprisingly, therefore, Java appears to be highly addictive to large numbers of software engineers. In Java's defense, it has, unlike older OOP approaches, recognized the importance of embracing the component model in addition to the object-oriented model, and thus one can find huge libraries of Java components on the market. Once again, as with the relational model, MitoSystems trod the Java path to see where it might lead when applied in the context of intelligence systems. We got burned. Twice! The truth is that higher levels of abstraction and productivity in software can only be attained by a corresponding higher level of abstraction for persistent data, and the integration of that abstraction into the programming metaphor. Object oriented languages, which lack this focus on persistence, are a false but shiny grail at the complex system level.

[5] Then there is the "lets just cobble together a bunch of different Commercial Off-The-Shelf ("COTS") applications" camp. On the surface of it, sounds like a good idea, right? Why re-invent the wheel when somebody has already made a wheel for you, and by putting together parts from a variety of vendors you can accomplish very complex things with a potentially huge savings in development cost. The problem here is that these people have failed to think of things in OODA loop terms. We all know how difficult it can be to get a piece of work product out of one application and into another without something being messed up. More importantly, each application, of course, has a different focus, interface and paradigm. Multiply these problems tenfold, and it soon becomes clear that we intend to overload our analysts exactly as they are right now, only this time we will give them loads of basically incompatible COTS applications to do it, instead of forcing them to access all our incompatible legacy stovepipe systems. COTS applications must, of course, be available, but they are poorly suited to closing an integrated OODA loop. Even if the analysts could handle it, do we truly believe that all this incompatibility wrestled to the ground somehow, is the way to keep an OODA loop spinning. What if we need changes? Are we going to get Microsoft to do them for us? Within a couple of days? Twice a week? In coordination with our other commercial application vendors? Hmmm…methinks this strategy needs a bit more thought.

[6] The Corporate Information Factory (CIF) approach has been proposed by some, especially within the usual vendors, as a potential solution for the problems at a system level. The CIF approach was invented by W.H Inmon around 1998. For a complete description of this methodology see "**Corporate Information Factory**" by W.H Inmon, Claudia Imhoff, and Ryan Sousa (Wiley 2001). CIF-based systems have had some considerable success when applied to corporate automation and knowledge systems, and it is perhaps these "knowledge" successes that led to the idea of applying this approach in the intelligence arena. But lets peel back the covers and see what is inside before we waste too much time getting excited. When we do this, we are (or should be) horrified to find that deep within is nothing more than a relational database. The CIF approach is basically just a formalism and set of procedures for how to set up and use your faithful old relational database, where to archive and warehouse the data, and how to create the various corporate applications that will need to access it. True, CIF acknowledges

the importance of historical data, and provides mechanisms whereby this can be accessed in a manner that is consistent with current data. True, also, that CIF acknowledges the diversity that exists between different types of system user (which mistakenly it then goes on to enumerate as a closed set). True, even, that it talks in terms of overlaying schema that convert the information within the system to the perspectives of the various corporate user types. Unfortunately, this is pretty much it. Still a relational database approach. Still not a whisper of OODA loops or how such a system might rapidly adapt in the face of changing requirements. Worst of all, what is it, and where do I buy one? The CIF is nothing more than a design philosophy; it is not a real thing, not a concrete architecture. Despite all the buzzwords, the only thing you can buy is a bunch of consulting time from people who have read the book and implemented past systems based on it. No, there is no solution to our problem here.

[7] The Associative Data Model is a new and innovative approach to the database problem. The associative model is a refinement of the binary model which first appeared as the entity-relationship model in 1976. Both models break the database content into two things: entities and associations. In the binary model, entities are related by associations. The associative model basically adds the ability for associations to be between entities, other associations, or any mixture thereof. For a full description of the associative data model see "**The Associative Model of Data**" by Simon Williams (Lazy Software 2002). As a matter of interest, this book contains some of the most eloquent and damming critiques around of the relational and object database approaches (and OOP for that matter), and is worth reading if only for these. A full discussion of the associative model's benefits, and there are many, is beyond the scope of this document. Here we must focus on its drawbacks. The associative model's main weakness is that it does not scale well. The reason is that the associative model, like the binary model before it, has abandoned the use of records. Every piece of every structure can be scattered anywhere on the disk and is accessed by reference. The opportunities for the kinds of optimizations necessary at very large scales are simply not there. It is this efficiency problem that has plagued the binary model, and the same charge can be leveled against the associative model. But regardless of this, the associative data model is simply a database metaphor. It says nothing about how we intend to actually design, code, and build our system. There is no associative architecture or environment. This final approach is in reality nothing more than "business as usual," but let us use a really "cool" database model.

4. How does Mitopia® Address the Overall Problem?

[1] We have seen from the discussions above that Mitopia® is a fully developed operational intelligence system architecture, and we have seen from our case study how it dramatically solves the problems of rapidly adapting to change (i.e., the IT system OODA loop) that tend to cripple other approaches. But how does Mitopia® address some of the other critical problems we have discussed, particularly that of facilitating the intelligence OODA loop?

[2] There are a large number of other problems that must be faced by intelligence architectures, and this author has written extensively elsewhere on these. However, here we are confining ourselves to just one problem, perhaps the most important problem, which is:

> **"We need a unifying community-wide architecture, capable of providing end-to-end support for the entire intelligence cycle from receiving requests from intelligence consumers through collection, analysis and back to dissemination to the intelligence consumers, and we need it now!"**

[3] We could double the size of this document by explaining all the nuances of what it takes to solve this problem. Suffice to say that Mitopia® is such an architecture; it is patented (12 in progress at this time); it is the result of 15 years of development; it is installed, and has been operational for more than five years; it is based on a number of now proven radical departures from current software thinking; it is designed to scale to the truly large (petabyte and above) with millions of users; it contains its own unique data model that is unlike any of those described above; it is rigidly based on a philosophy designed to make it adaptive; it provides for user level configuration and changes; it provides a published API containing many thousands of calls allowing complete customization at all levels; it handles distributed, clustered, and federated servers; it handles multi-lingual issues; it has an integrated GIS and visualizer suite; it supports all phases of the cycle from equipment control (for data capture) through to multimedia report generation and presentation; …and on and on. Mitopia® is available right now, can be installed in months, and is entirely developed in the U.S. MitoSystems is a wholly U.S. owned corporation. We understand the problem; we've already built a solution.

[4] As a service to the poor rabbit family frozen in the roadway, the author offers the following minimal checklist for evaluating any proposed intelligence system architecture. To be worth even spending time looking at, a proposed system must score at least 75% positive responses overall and 100% on all items in the must-have section. For comparison, the Mitopia® architecture scores 93% overall on this test, and 100% on the must-have section.

Intelligence Architecture Evaluation Form

Yes/No

MUST HAVE

- [ ] Is not based on the relational database model
- [ ] Has thoroughly addressed its own OODA loop problem
- [ ] Can be changed by customer staff, not the original vendor
- [ ] Has integrated GIS
- [ ] Has addressed issues of scaling, federation, distribution, and clustering
- [ ] Provides automated user-definable alerts
- [ ] Supports the entire cycle from collection to delivery to the consumer
- [ ] Is a knowledge-level (i.e., ontology-based) system

Yes/No

SHOULD HAVE

- [ ] Exists in some form right now and can be demonstrated
- [ ] Handles multilingual issues and cross-language searching
- [ ] Is data-flow, not control-flow based
- [ ] Has run-time alterable/accessible type substrate
- [ ] Has integrated visualizers
- [ ] Handles multimedia content including video, sound, and images
- [ ] Can easily integrate and control other COTS applications
- [ ] Handles units and unit conversion
- [ ] Has a visual programming language for component assembly by analysts
- [ ] Provides for configurable security protocols and gateways
- [ ] Supports a 'flat' data model for rapid data 'flow'
- [ ] Supports workgroup collaboration
- [ ] Provides support for user configuration of the interface
- [ ] Easy integration/ingestion of legacy systems
- [ ] Supports inter-organization communication and ontology gateways
- [ ] Contains built-in integrated support for robotic mass storage subsystems
- [ ] Contains inverted-file support as part of its data model
- [ ] Provides large, published code-level API for customization by others
- [ ] Supports customization of server behaviors, not just client behaviors
- [ ] Provides user-centric on-the-fly hyper-linking domains

Form:BEEF-UBI-EST

5. Summary

[1] To summarize our comparative findings between a Mitopia® based approach and a conventional approach using on our case study, they are as follows:

(a) We have reduced the initial system development and installation time by a factor of 4 from 8 years to just 2.

(b) We have reduced the initial system programming development cost by a factor of 25 from $160 million to just $6 million.

(c) We have reduced on-going maintenance costs by a factor of 50 from $25 million/year to just $500K per year.

(d) We have dramatically reduced, perhaps eliminated, the risk to the customer of failure to successfully develop such a system. More importantly, we have reduced the time that the customer must wait to see if his system stands any chance of meeting his requirements from 5 years to just a few short months. The value in lost opportunity cost of this 5-year head start to the customer organization, and those that rely on it, is difficult to overestimate.

(e) We have improved the OODA loop of the customer software system, and the customer organization itself. In the case of the system OODA loop, we can see from above that this factor is something between 50 and 100 times.

(f) The delivered system is based on a proven and evolving architecture that is also shared by others. Our ability to interact with other Mitopia®-based organizations (a feature which a standard approach is unlikely to incorporate) is now guaranteed. Moreover, the system incorporates a whole host of features that we will subsequently find that we need, even though we would be unlikely to put them in the initial system requirements for fear of causing the project to fail. The system from the outset (i.e., in the first few months after development starts) has the ability to handle multi-lingual issues, it incorporates rich multimedia content, contains built-in visualizers for complex data analysis, has an integrated GIS system, handles the issues of globally distributed, federated, and clustered servers to ensure that it can scale, contains built-in support for robotic mass storage archives, can easily integrate with legacy systems, contains a built-in data-flow based visual programming language to allow system evolution by non-technical users such as analysts, provides automated and continuous warnings and alerts, and supports the entire intelligence cycle from collection through to interactive delivery to the decision maker. These are just a few of the integrated features provided by the basic Mitopia® architecture and they require no programming effort from the system-developer to incorporate. Most importantly, above all, it is an intelligence *platform*, not an application. We can now benefit from all new developments by others both within MitoSystems (for

the Mitopia® architecture), and from without, as customers and external developers create an expanding set of analytical tools and functionality that can be easily plugged into the system via the visual programming environment, and then used to facilitate or automate system specific tasks. The system is free to evolve rapidly as needs and requirements change.

[2] From the discussion above, it is clear that we need a radically different approach to solving the problem of intelligence systems. A practical compliant architecture must be based on the concept of a distributed data-flow driven environment, rather than a conventional control-flow based solution. The form, content, and behavior of the data in the environment must be described via an ontology that is both specific to the given application, and general enough to allow efficient communications across organizational barriers. In such an environment, control- or data-flow based programs must begin execution by virtue of a matching set of data objects or tokens appearing on the input data-flow pins of the program. When programs complete, they produce a set of resultant data tokens on their outputs that then become part of the environment (persistent or otherwise). By contrast, conventional systems allocate execution time to a program without knowledge of what it is actually doing, and it is up to the program itself to seek out and acquire its required inputs. To do this, the program requires detailed knowledge of its environment, and the need for this knowledge reduces the generality of the program, and increases the overall rigidity of the system, thus making it resistive to change, and more likely to develop a "stovepipe" topology. Unless we adopt a radical approach to attacking the problem like that described in this paper, there is little or no chance of creating any workable unifying intelligence architecture within the foreseeable future.

[3] If the Federal Acquisition Requirements ("FARs") are the way we intend to specify the software systems that the government acquires, then these need to be modified or extended to require all vendors to provide an OODA loop analysis, similar to that given in the case study above. We must stop ignoring change in our designs. It is inevitable.

[4] The Mitopia® architecture described in this paper is the only generalized, fully developed, installed, and long-term operational system that the author is aware of, that comes close to systematically addressing the problems described in this document (and a host of others not detailed herein). As such, it is probably the only viable candidate at this time on which to attempt to build future intelligence systems to address the deepening problems we now face. Into this architecture we can tightly integrate the many valuable technologies that others can provide to address specific portions of the problem. By permitting all information players (large and the small) to profitably contribute in a well-defined manner on top of such a unifying architecture without the infrastructural barriers they now face, we can collectively and rapidly evolve our way out of our problems. If other such architectures exist at present, then we should look closely at them, and use them if they are operational and can be demonstrated to be superior. The only thing we cannot responsibly do is to continue to ignore the problem. Time is running out.

[5] The dinosaurs ruled the earth for 120 million years. They are all gone now. Why? Because they failed to adapt to change. The mammals were around at the same time as the dinosaurs and they/we now rule. They/we adapted. In the information world, timeframes are infinitely shorter and change immeasurably faster. Our intelligence infrastructure has ruled unchallenged since the 1940's, but the world is changing. We must recognize our problems, and learn to adapt, or we are destined to tread the same path as the dinosaurs.