

## **Appendix R**

# **Lessons-Learned from the BSY-2's Trenches**

---

## Content

<b>R.1 Introduction</b> .....	R-3
<b>R.2 People</b> .....	R-3
R.2.1 Motivation .....	R-4
R.2.2 Change Is Good .....	R-5
R.2.3 Diversity .....	R-7
R.2.4 Software Architects .....	R-9
<b>R.3 Process</b> .....	R-11
R.3.1 Initial Establishment of a Process .....	R-11
R.3.2 Interfaces .....	R-12
R.3.2.1 Utilize Code in the IDD .....	R-12
R.3.2.2 Put Inter-Process Interfaces in the IDD .....	R-13
R.3.2.3 Keep SRS Interface References to the Message Level .....	R-13
R.3.2.4 Combine the IRS with the IDD .....	R-13
R.3.2.5 Configuration Management .....	R-13
R.3.2.6 Distributed Object Systems .....	R-14
R.3.3 Integrated Product Teams .....	R-14
R.3.4 Reuse .....	R-15
R.3.5 Waterfall versus Spiral Model .....	R-16
R.3.6 Configuration Management .....	R-17
<b>R.4 Technology</b> .....	R-18
R.4.1 Use of Ada .....	R-18
R.4.1.1 There's No Substitute for Experience .....	R-18
R.4.1.2 Self Documenting Code .....	R-18
R.4.1.3 Use Tasks Wisely .....	R-19
R.4.1.4 Integration Techniques .....	R-19
R.4.1.5 Design for Tuning Up Front .....	R-19
R.4.1.6 Ada Is Not for Everything .....	R-19
<b>R.5 Summary</b> .....	R-20
<b>R.6 References</b> .....	R-21
<b>R.7 About the Author</b> .....	R-21

**Robert F. Sullivan Jr.**

PROSOFT, Inc.

---

## R.1 Introduction

You've read the headlines. You've watched "60 minutes." Large government software development programs ending up in huge cost overruns, way over schedule, and no clear indication of either the quality or the functionality that the government is receiving for the enormous price tag. Everyone wonders what went wrong. So when a subset of the seven sensor AN/BSY-2 Combat System (the single sensor AN/BQG-5) was deployed on the USS Augusta in 1994, and initialized properly the first time it was powered up on the submarine, and generally performed outstanding in its first sea trials, you have to start asking what went right. The following areas are arguably the most succinct summary of what went right. However, there are a great many lessons learned. These 4 key areas are not a silver bullet. The areas are:

- Emotional Mission Statement. There was a strong, emotional purpose attached to the project and everyone involved bought into it;
- Process improvement culture. Over the course of the project, there were vast, and essential process improvements;
- Strong configuration management. Reliable, effective, automated baseline control and problem reporting;
- Use of Ada. Strong typing, information hiding, and other factors lead to a solid, reliable product.

To get an appreciation of BSY-2 and the magnitude of its success, you have to appreciate the immense size of it. It is over 3 million lines of tactical code, with several million lines of support software. The tactical code is broken up into over 100 CSCIs. Most CSCIs were assigned to an individual team. Several of the CSCIs are so large and complex, no one person knows all the details the CSCI. Needless to say, no one person knows all the details of the entire system.

For sure, there is some breakthrough technology involved. But the challenge of creating that technology is dwarfed in comparison with the challenge of creating the countless pieces of "trivial" functionality that must all play together. Taken individually, most of the pieces are relatively easily understood. But that's the danger and the ultimate challenge. Few of the pieces can be taken individually. They all must play together with other pieces. The inter-team interaction required to pull this off is unlike any other program. Its difficult enough to balance the dynamics within one team. Imagine trying to balance the dynamics of over 100 interacting teams? Imagine if the teams are spread over several geographic locations? Imagine if not all the people on a given team report to the same functional management? Imagine if the support groups like configuration management, test support software, simulation/stimulation software, and CSCI integration report to different management than the majority of organizations they serve? Sounds like a formula for disaster? Well it could have been. The successful techniques can be broken up into management of 3 categories: people, process, and technology.

---

## R.2 People

There is little breakthrough technology involved, so we aren't talking about finding a few superstars, giving them everything they want, and watching them do good things. We are talking about finding scores of solid, team players. This challenge is more difficult than any technology problems faced. You need to find good people, properly reward and motivate them, ensure a dynamic organization that responds well to change, and balance inter-team dynamics.

---

## R.2.1 Motivation

**Create an emotional mission statement that achieves buy-in from the entire team and their families.**

This project required a tremendous, sustained commitment for many years. With anything this size, there will be built in bureaucracy, to the point where it's hard to do the right thing; much easier to conform to inefficient rules and regulations. But conformation leads to stagnation and over the 7 years, there are bound to be profoundly better ways of doing business.

How do you get several hundred people motivated to sustain long hours away from their families and friends, work crazy hours and keep bucking the system to make it better, no matter how hard it is? Just like you'll see in many "peak performance" books, the trick to superhuman effort, be it individual or team (although much more vital to a team), is commitment to an important mission. Everyone needs to feel important and adequate. What better way than to contribute to something vital to our nation's security? The Seawolf computer system!!! The Soviet Union had caught us sleeping, had advanced their submarine technology beyond ours and we needed to catch up in a hurry.

*What a motivator!* Everyone knew the importance of their contribution. Everyone wanted nothing but the best in quality. No one would settle for anything less. The customer didn't need to worry about getting shortchanged. There was a passion for quality, a passion to build the best that America had to offer, the best in the world. One bad apple couldn't spoil the show, a dozen others would expose and correct the problem and preserve the integrity of the project. A tremendous emotional motivator produced superhuman results.

Any motivator such as this must be emotional. It has to touch the families of the engineers, not just the engineers. There is a sacrifice on everyone's part, and everyone must know what they are getting for that sacrifice. Peace of mind, the Seawolf sub will protect us from the bad guys. It will make the world safe for democracy and freedom. And we would prosper as a company with it. We have found that financial rewards are a side effect of a good emotional motivator. It is much more effective than a financial motivator, alone.

Print up T-shirts, get coffee mugs and hats with exciting, emotional logos and pictures. Outfit the family with the paraphernalia. Have family events and tours of the facilities. One of the most difficult aspects of this job was the security. We were sequestered off in a closed room. No family members allowed. It was 6 years into the program before I thought to bring home the office seating diagram. My wife was elated because she could relate better to my world. A simple little picture gave her a whole new understanding and appreciation for my work. And it increased her buy-in.

Watch the results. If the family buys into the mission statement, you've got it made. Tremendous things will happen. If the family doesn't buy in, you've got an uphill battle.

So how did we keep going when the Soviet Union collapsed and removed the threat, thus removing the main point of the mission? It's never been the same. But that doesn't mean it demoralized everyone. Sure, there were a few rough months. Sure, many good people left. Sure, some people retired on the job. Then throw on top of that the congressional budget cuts, eliminating production of several systems per year to only 2 systems overall!!! Not only is the mission removed, but the economic stability is damaged. Family security is at risk. How can you recover from something like this?

We did the only thing we could. We found another mission. We would strive to make the Seawolf technology be the foundation for the New Attack Submarine (NAS). The NAS is vitally needed for trouble areas like the Persian Gulf, where our current subs are too big or not sensitive enough. It wasn't a great substitute, but it was the only option available. It revitalized many and gave the promise of economic stability and growth.

We were back on track. Until the NAS funding got delayed, pending the success of the Seawolf. Now the pressure was on. Our economic stability was tied to the completion of this program, and even then, there was no guarantee. There might be a delay between the end of the Seawolf and any NAS funding. The family security was damaged. The mission was weakened. Luckily, there was enough momentum and enough process safeguards that the project will complete no matter what. Had this happened a few years ago, we would have been another disaster on 60 minutes.

*In summary, if you don't have an emotional mission, get one.*

---

## R.2.2 Change Is Good

**Create a culture where constant change is encouraged, embraced, and rewarded.**

People tend to resist change. It makes them uncomfortable. It's harder because you have to learn a new way of doing business. It worked this way before, why can't we stick with it?

With a program this size, most of the old rules are history. With the global competition, most of the old rules are history. One management expert describes the only true, sustainable, competitive edge, as the ability to learn faster than the competition.[1] Learning new things implies changing the way you do business. On a program that stretches over 7 years, there are going to be vast changes in industrial best practices. If you don't prepare for change, it will hurt. Either you won't change and become obsolete, or the changes will not be embraced properly and you won't get the payoff.

How do you prepare for change? Well, it's got to start with upper management, and end with everyone involved. It's got to be embraced by everyone, including (and especially) the customer. The typical DoD project gets its share of standards imposed on it (including the contract itself). However, these standards are made to be tailored to industry and the contractor's best practices. It is in everyone's best interest to do this. Blindly adhering to a standard is no excuse for thinking, adapting, and changing. You and your customer should approach each standard as if its first requirement read, "You shall continuously refine, document, and improve your interpretation and implementation of this standard". Associating change with a "shall" makes it a requirement that needs to be met.

What happens too often is the customer and the contractor's standards enforcers, commonly known as gatekeepers or speed bumps, enforce the standards for the sake of enforcing the standards. There is no room for interpretation or change. There is often no written interpretation either. This mindset is deadly. It leads to demoralization and resentment. It stifles creativity. It creates bottlenecks with no apparent value added. We'll talk more about the process specific aspects of this later but let's concentrate on the people aspects now.

You've heard of manufacturing success stories where direct discussions with people doing the work, senior management, and the customer produced miracles. Software development is no different. If change and improvement are part of the contract, part of the requirements, and embraced by everyone involved (especially

the people in the trenches), people will create better solutions than any standard could have dictated or even hoped for. There is no standard process or guideline that should be imposed blindly on any software project, let alone imposed across the board on a huge project. Once a standard is in place, naturally it should be documented and enforced to the extent practical. But if the culture is not in place to constantly refine and improve standards and their implementation (yielding more efficiency without a compromise in quality), you will waste big bucks, and build frustration and resentment.

With the refinement must come the rewards. Now there are several lessons here. First, the struggle to implement change is often made unnecessarily difficult due to the bureaucracy, or inertia built up. Without the culture described above, you often end up with these brick walls on both sides — contractor and customer are both afraid to change. The walls are protected by the standard's enforcers (gatekeepers). Interpretation and implementation is often dictated or heavily influenced by the gatekeepers. The engineers implementing the standard are often unequally represented, both internally and with discussions with the customer. The fear is that the engineers will take the shortest path, thus compromising quality and product integrity. But in reality, rigid adherence to any standard cannot, by itself, ensure a quality product. Yet for any standard or process to be effective, the people doing the work need to buy into the spirit of the standard.

This apparent deadlock can be broken by tapping into the natural tendency of many (some argue all) engineers — they like to complain. They are constantly striving to avoid the hard stuff. But that is the irony. That is what they are trained to do, to create products and services that make tasks convenient, easy, or efficient. Without problems to complain about, they have no mission (there's that word again). Encourage and develop their complaining skills, and demand that they do something about it (active complaining). This is a reward in itself. By having more control of their own destiny, just about anyone is bound to have an increase in morale.

In this new style organization, the gatekeepers (for both contractor and customer) are still a vital piece of the puzzle. After all, a standard or process is nothing if it is not followed. The typical gatekeeper is very knowledgeable in the vast myriad of standards. As a change is proposed, the new role of the gatekeeper should be to ensure continued compliance with the contract (including standards imposed by the contract). Embrace the change as something that will make the standard, and consequently the product, better, not as something that threatens it. When a change causes a requirement (or interpretation of a requirement) in the contract to change, they should participate in the modification of the contract.

A very effective vehicle to accomplish this is a “Memorandum of Agreement” or a “Memorandum of Understanding.” These are nothing more than a change to the contract or an interpretation or clarification of the contract. However, it is drafted with the help of the people implementing it. Since it is drafted by the people charged with implementing it, by default it carries a much higher probability of success than one drafted by some authority. This is not meant to be a dissertation on how to coddle an engineer. Complaining must be measured against productive change. Changes must be weighed against programmatic (cost and schedule), contractual, and technical objectives. But there is often tremendous room for flexibility within these constraints, especially over a long development cycle. With this flexibility will come improved morale.

Now all pieces described here are vital to a creating successful culture of change. The people implementing a standard must be equally represented on all decisions relating to that standard. The people enforcing the standard must know and embrace their role. The customer must encourage change by the contractor. And management must reward change when it pays off. One final note relates to an observation and suggestion on an ideal organization to implement change. Most organizations, once they establish a process, establish some form of process improvement team. This can range from informal to formal. In either case, there are

often strong individual advocates for change that get involved one way or another. But even the most staunch advocates for change are susceptible to building walls and impeding change. Advocates for change often fight passionately about a few key issues that are close to their heart, often after they have spent many frustrating hours struggling with the current system. They typically can not sustain the energy necessary to constantly improve and question other aspects of the system.

To solve this, I suggest an organization where the improvement team is constantly changing also. This can be “chaired” by the same person but the people doing the leg work need to have fresh legs. By making the team’s accomplishments recognized, involvement on this team becomes a reward, an opportunity. You will get plenty of fresh legs asking to get a chance to play. The veterans are still required to make sure the integrity of the product is maintained. But by constantly getting fresh legs, a fresh look at existing problems, you foster more innovation. And the competition can instill renewed vigor in the veterans.

*In summary, change is not only good, it is required.*

---

### R.2.3 Diversity

**You need to motivate as many people as possible, to be as productive and happy as possible.**

With any large program, you will have a melting pot of people. From self starters, to superstars, to lazy but effective (if properly motivated) people, you’ll see them all. The trick is to get as many people as possible, as productive and happy as possible. This involves quite the juggling act when there are scores of engineers, each with their own aspirations and needs. The corporate ladder becomes harder and harder to climb. Much of the management and senior technical people are entrenched in their positions, and necessarily so for consistency and continuity to the program. After all, we are talking about many years of development.

This is an eternity for a project. What’s in it for the average engineer? Where’s the career path, the upward mobility? Sure, there will be some attrition in the management and senior technical chain but there are dozens waiting to fill those shoes. The key becomes making each contributor feel like a vital piece in the big picture. Software development is a team sport. On a large program, it becomes a multiple team sport. You can’t rely on a few superstars to carry the whole load. Upper management needs to balance the importance of each team. Middle management or the team leader needs to balance the importance of individuals on each team. As a project matures, different teams play different roles. At one time in the program, a certain team may be facing the number 1 high risk issue on the program. This team will get more than its share of attention. If the team performs well, it will get some rewards. The balance of criticality, performance, and rewards, as perceived by other teams, is the most important factor in keeping a diverse project productive.

This balance requires a positive environment. On a project where it is so difficult and long, it is easy to get caught up in the practice of catching people doing something wrong. After all, we all need to look for problems to solve, we all need to constantly improve to stay competitive. But the masses will thrive in a culture that catches people doing something right, and rewards those positive actions. The typical rewards just don’t cut it on a large program. Most organizations will have some form of management awards, some will also have peer awards. Certainly, these and salary actions are the most tangible rewards. But this is only the tip of the iceberg. The rewards must be as diverse as the people involved. Each person has attached their own emotional reasons for their commitment to the mission. Understand that reason, each person’s motivation, and you have the key to the proper rewards.

For example, some people are motivated by a purely technical challenge. This person would be thrilled to receive a gift certificate to a local computer bookstore, or a personal Internet account, or attendance to an industry conference. Others are motivated by family security. What about a family oriented reward? Like a dinner certificate for the entire family? The late hours wouldn't hurt so bad, the spouse and kids would think the company really appreciated the whole family's commitment. Another example is someone with many leisure activities. Some unexpected paid time off after a few long months would make it easier to work the overtime when it is called for again (and you know it will be called for again, and again, and again). The rewards don't need to cost money to be effective. A visit by someone from upper management to a team meeting before, during, or after a big deadline to say thanks for the effort goes a long, long way towards instilling a feeling of appreciation.

Earlier we talked about perception. This is nine-tenths of the law. The perception of balance is more important than the reality of balance. Not all jobs are glorious. Not all teams get the exciting work. Not all teams are equal (nor should they be). But all jobs and teams are vital. All teams must feel vital. It may take some creative advertising to increase the perceived criticality of some tasks, and the team that performs those tasks. Equally important, if a team doesn't have the reputation for performance yet gets recognized for an achievement, resentment will grow. And obviously no one team should get all the glorious work. Balance the perception between teams and you are half way there.

If you don't achieve this type of balance, you will fail in other inter-team aspects. All teams will need to interrelate to other teams in some way. The products produced by one team may need to be used by another team (e.g., a document produced by the Systems Engineering Team needs to be used as the requirements for building the product by the Software Engineering Team). Personnel may need to be reallocated from one team to another. If the teams aren't balanced, you will see negative side affects in the team members, and in the team leaders (e.g., middle management). This will produce an impossible situation to deal with for upper management. Now we aren't saying make everything equal. Balance does not imply equality. Some jobs are more difficult and will need better players. Some jobs are less difficult and can get by with lesser skilled players. The point is the differences in skill level needs to be accepted, and respected, not flaunted. Below are some examples.

One of the main reasons for poor proliferation of reusable software components is the "not invented here" or NIH syndrome. Inter-team conflicts reinforce NIH. We have seen reuse fail and we have seen it work. In all cases, it can be largely attributed to poor or good team dynamics. In one of the failures, the people doing the work did not have a strong reputation for technical abilities. The result was the other teams had an inherent resistance to adopting anything produced by this team. Contrary, in one of the best success stories, some of the best technical people of two teams were pooled together to create a reusable application architecture. Because the right people were chosen and their abilities were respected (not flaunted), adoption of the architecture was not questioned.

On any project, there will be changes in the needs of (people) resources for all teams. On a huge project, any given person will likely work on several different teams over the life of the project. If there are inter-team conflicts, it will be reflected in the middle management. They will start to protect "their" best players to keep their team strong, rather than to let the best person fill a particular need on another team. The result is the whole project suffers. You can work around weak links in the system but eventually these weak links will reflect on the entire project. You've got to be able to "repair" or help the weaker links by improving that team. Sometimes this demands drafting someone from another team. If there is any conflict between teams, human nature will often keep management from sacrificing on his or her own team rather than doing the right thing for the good of the entire project.

Ironically, when the manager has been willing to sacrifice a key player for the good of the entire project, that player (and most others on the team) feel deeply committed to that manager. They tend to be willing to go out on loan, as long as they can stay functionally assigned to that manager. So the unselfish manager gets the best of both worlds he or she helps the program and also earns tremendous loyalty and respect from the players.

Another classic example is the typical “advisor” organization, or staff engineers. This is supposed to be the group of experts and hot-shots. Their charter is to provide advice and guidance, and to set policy and direction. Think about the resentment this organization will face if the people in this group are not perceived as experts? Or if the people in this group are arrogant, or condescending. Who would go to them for help and advice? This type of group is perceived as a service group. As with any service group, the customer (in this case, the other teams) must come first. Service must be with a smile. All actions by this group must show tangible benefits to the teams they serve. What you are trying to do is get other teams to learn from this group. Respect, integrity, and benefits are the key ingredients. Get your service teams to exhibit these characteristics and watch the results.

*In summary, balance the perception (of criticality and rewards) between teams, then balance the perception within each team.*

---

## R.2.4 Software Architects

**Like a building, a software system needs a solid architecture.**

In analyzing an individual team, or more importantly, a collection of tightly coupled teams, it is helpful to use Dr. Covey’s jungle warfare model.[2] There are the people wielding the machetes (coders), people sharpening the machetes and motivating the wielders (managers), and the lookout up in the tallest tree directing the energy of the machete wielders, guiding the path, making sure they are in the correct jungle (software architect). To this model, we would like to add the policy makers, the ones that put the team in the jungle in the first place with some objective (systems engineers). Without a balance in all parts, a team will lose effectiveness. If one team within the entire project loses effectiveness, other teams suffer and need to compensate.

Our experience has been that the most difficult role to fill is the lookout, the team leader, what we call a software architect. A software architect is a good communicator, an effective system engineer, an interface expert, and a proficient software designer. Often the ideal software architect can pound out excellent code and a typical reaction is to keep that person producing. This is a mistake. Many people can develop into proficient or adequate coders. Not everyone can become a proficient designer. Fewer still can become interface experts. Even less can understand requirements, the big picture. Rare is the individual with all the other attributes plus the ability to communicate.

These software architects must be carefully detected, and removed from a heavy production role. They are much more effective in this new role. On a large project, software architects are needed to provide the glue to hold the entire system together, especially in the early phases. It is difficult to document and articulate precisely a software architecture. It is more of a concept than a collection of rules. Until the foundation of the system is matured, the software architects must provide guidance and direction to ensure that all the pieces will fit together.

This is where interface definition and communication skills are most vital. Interfaces are the biggest (and sometimes the only) inter-team communication method. The combination of the CSCI architecture and the interface definition becomes the architect's leverage and communication vehicle. In designing interfaces, data format and content is important, but message sequencing in all phases of a CSCI's life are more tightly coupled to a CSCI's architecture. The sequence of messages must take into account initialization, reconfiguration, failures, and restoration. The CSCI's tasking interaction model, input and output mechanisms and sequencing, and major state sequencing (e.g., initialization, reconfiguration, etc.) all make up its architecture. Understanding the true requirements is essential to this task. As you can see, this is a rare individual.

Software architects are not always easy to spot, although there are trends. There seems to be an inbred attitude to be unselfish, to sense the greater need, to work on what's really important, not what's urgent. They are compelled to climb the tree to get a look (at the big picture), rather than to attack every clump of brush (code) they see. However, since they can produce, if pressured, they will produce, and often with star or superstar results.

Here is the danger. It is very difficult to comprehend that your best coder, who also happens to be your best (and quite possibly your only) architect, should be taken off coding and put on requirements analysis, interface design, software tasking interaction, and design guidelines. Why can't he or she just review other peoples design or code to make sure it is close enough, while he or she keeps swinging the machete? Why have the lookout direct the energy of the machetes before its expended? Energy spent is not recoverable. Development dollars spent are not recoverable. This is your most precious resource. And a good software architect can give you the most effective balance.

How? By making sure the vital tasks are done with consistency and accuracy. Requirements analysis is accepted as vital. But different pieces within a CSCI are often reviewed by different people. Often designers or coders are given a piece of the CSCI and allowed to run with the entire piece, from requirements analysis to tasking structure and software design, to coding and integration. The results can range from excessive use of tasking, to incompatible interfaces, to correct implementation of the requirements but discovery that the requirements had some fundamental flaw. This cost is compounded in a large system. The results often affect another team. Expand the problem from a stand-alone functional piece of one CSCI to a more complex scenario, the requirements for a thread of functionality that spans several CSCIs. Without a software architect ensuring that the thread will work from a structural and architectural point of view, its not worth getting the other players in the game yet.

Good team dynamics are essential to effectively utilizing a software architect. The architect doesn't typically write requirements, but guides their writing, ensures that each requirement fits in with the vision for the architecture. This can cause problems if the architect and systems engineers don't share mutual respect. It is even more difficult if the systems engineer is part of a different functional organization. On the positive side, establishing the position of software architect frees up a lead design spot for an individual CSCI. This gives other engineers a growth position and a chance to learn from the architect.

Some CSCIs need a software architect all to themselves, while in other cases one architect best serves a collection of interrelated CSCIs. Interaction between CSCIs is the most important aspect to manage early, and the most chain reactive. With a large program and the typical matrix organization, it is inevitable that the ideal coupling of CSCIs to architect will cross some organizational boundaries. Tough. If you don't do it, you will pay many times over. Give up your architect for the good of the project. Give the architect the authority to influence your CSCI's structure, to be consistent with the rest of the group.

*In summary, identify and support Software Architects. It is a key ingredient to success.*

---

## R.3 Process

Effective use of a process was one of the most important lessons learned. We define a process as the set of procedures used to produce some result. Thus, process affects every aspect of the development. The key areas presented here are the initial establishment of a process, several life cycle specific issues, the waterfall versus spiral (or evolutionary) development model, and configuration management.

---

### R.3.1 Initial Establishment of a Process

**Lets figure out how we want to do business, before we do business.**

Initial establishment and continuous improvement of a software development process is an investment and an attitude. You need to assign some of your best people to defining it (investment). One of the biggest dangers is letting who ever is available work on process. The group defining and improving the process is viewed as a service group. Remember the lessons about team dynamics. The service group must command respect and exhibit integrity. The group must provide demonstrable benefits. By putting some well respected individuals on this team, you start off on the right foot.

Initial establishment is often one of the most difficult hurdles in any organization. If there has been no formal process, it is difficult to comprehend the need for one (attitude). This is compounded when tackling a project that dwarfs any other project performed by that organization. It is difficult to anticipate just how different this large project will be. The need for a solid process that is embraced by all grows exponentially with project size. Obviously you need to change the attitude if you want the process to pay dividends. If the leaders have bought into the need for a process, and participate in its creation and maintenance, the rest will follow and watch for results (which must come soon).

Here are some tips on how to start. First, get complete buy in from your leaders. This must eventually include all groups that will be touched by the process, from the program office and systems engineering, to software design, test, integration, quality assurance, configuration management, and last, but not least, the customer. Start with a small group of representative people that can work together. It doesn't matter that you might be leaving someone important out yet, getting the ball rolling is most important at this stage. The people that will eventually use this process need to see momentum, not promises.

Start out simple and flexible. Don't take the whole life cycle at once. Grow the process. As you tackle different pieces of the life cycle, add the appropriate organizations to the team. One size will never fit all so don't try to spell out all possible combinations. Rather, strive for the architecture of the process. Let each team or subsystem tailor the process for their specific applications. Here is where the software architects can play a key role. You don't want each team to do their own thing, because that cancels out the consistency across the board, which is important for efficiency in the support organizations and to the customer. Try to keep each team or subsystem's tailoring definition to one or two pages.

Strive for readability first, completeness second. It doesn't matter if the process is 100% complete if no one reads it, follows it, or checks for compliance. Challenge each step to calculate the added value. Value is a difficult quantity to measure or anticipate. Perception plays a strong role here. If everyone perceives a certain step adds value, they will attempt to follow it rigorously. As discussed previously, rotate people on the process improvement teams to keep fresh ideas coming.

One final note on participation. Similar to the team dynamics lessons, all groups that touch the product in some way, shape, or form should be represented in the process and on the process definition / improvement team. If one group is excluded (by their or someone else's choice), problems will arise. Resentment will build. Quality will suffer. You need complete participation.

*In summary, process is an investment and an attitude.*

---

## R.3.2 Interfaces

### **A large system lives or dies by its interfaces.**

Interface definition is not part of the classical life cycle. Yet interface definition on a large, distributed system is vital. This can not be stressed enough. The problem is compounded when the development team is geographically separated. Concurrent development and the typical waterfall mentality add further complications.

Typical implementations of MIL-STD-2167A call for an IRS (Interface Requirements Document) and an IDD (Interface Design Document). The way I look at the interfaces is like a contract. If I sign my CSCI up to an interface, I am signing a contract that I will hold up my end of the deal. I am assuming the other side of the interface will do the same. It is an excellent vehicle to manage parallel development. However, it takes quite an effort to completely define one CSCI's interfaces. It is not a waterfall type activity. It can not be completely specified up front, any more than the requirements for a CSCI can be completely specified before designing or coding anything. The amount of detail required to unambiguously specify an interface is tremendous. Here are some tips for success. As discussed previously, the software architects should play the lead role in managing interfaces.

### **R.3.2.1 Utilize Code in the IDD**

The use of code in the IDD, to the extent possible, provides an unambiguous definition. The use of Ada allows complete message formats to be specified. But this is the easy part of interface definition. The format of a message doesn't impact the CSCI architecture as much as the communications address of the message, sequencing, frequency of transmission, initiating conditions, and expected responses. These latter interface requirements can have a direct effect on a CSCI's tasking structure, its architecture. Some of these items may be well represented in code. For example, the frequency of transmission can be a constant. The initiating conditions and expected responses could often be specified as the identifier of another message (e.g., this message is initiated upon receipt of message X). As we'll discuss later in the section about reusable CSCI architectures, this type of information can go a long way toward removing any ambiguity from interfaces. The more information that is unambiguous, the more effective parallel development will be, and the easier integration will be.

For Ada to Ada interfaces, as long as both sides utilize the same compiler, the definition is sound. If the interface is from an Ada to non-Ada CSCI, it isn't obvious how to proceed. The Ada CSCI needs the definition to be in Ada anyway, so as a minimum, that work needs to be done. Here is where I would recommend the generation or purchase of a tool. This tool would take the Ada interface definition and generate the non-Ada definition. In the case of an assembler based CSCI, the complete definition could be specified using "equate" statements. This type of convention would need to be specified at the start of a program.

### R.3.2.2 Put Inter-Process Interfaces in the IDD

The larger the CSCI, the larger the need for some type of internal interface definitions. This does not need to be formal, as in an IDD. But we have seen large programs with few CSCIs struggle due to a lack of internal interface control. In these programs, large CSCIs may physically be spread over several processors and/or processes. These CSCIs could be better managed by either splitting the CSCI (across physical boundaries, i.e., processor or process boundaries, or individual Ada programs), thereby causing more information to be put into the IDD, or by documenting internal CSCI interfaces (again, using physical boundaries) in the IDD. Putting these internal interfaces in the IDD causes some benefits. First, it raises the importance of the interface. It won't be carelessly changed. Second, it allows more parallel development by establishing a contract between the different pieces of the CSCI. Third, it benefits integration. By putting these interfaces in the IDD, it allows test software, simulation / stimulation software, and data collection software to utilize the same interfaces as the CSCI under test.

Typical embedded systems often have one application process per processor, but as future systems move to a COTS (commercial-off-the-shelf) hardware environment, many processes may co-reside on the same workstation. This increases the need to document these interfaces in a formal manner.

### R.3.2.3 Keep SRS Interface References to the Message Level

We've seen a lot of SRSs, and we argue that they could all adequately specify the requirements by referencing only the message level. The key things needed within an SRS are the processing and sequencing around the messages. What happens before sending this message, what should we do when we receive this message. The algorithmic details in the SRS may imply some data format content but refrain from putting that detail in the SRS.

### R.3.2.4 Combine the IRS with the IDD

All SRS messages must be represented, one for one, in the IDD. Since the SRS doesn't mention message content, only message name and sequencing/processing requirements, the IDD is still free to implement that message using any style necessary. On a distributed system, each CSCI under test will need to be stimulated using the IDD anyway. The IRS interfaces are of no use. The requirements in the SRS for message sequencing are utilized to generate test cases for formal CSCI testing. The message content details from the IDD are used to create the test case messages.

Within the IDD, all sequencing details for SRS specified messages become requirements. All message content becomes implementation. Implementation only messages are still allowed in the IDD. These messages are typically utilized for low level handshaking that is not appropriate for an SRS. Since the IDD contains as much code as possible, including the complete message format, it gives you binding power and a built in management capability. The IRS is paper or an electronic model, and has no binding power. Therefore, it will inevitably drift out of date. If everyone is forced to use the same Configuration Managed IDD when building their code, interface problems will tend to settle themselves.

### R.3.2.5 Configuration Management

Once an interface agreement is reached, get that agreement into the formal program baseline as soon as possible. Ideally, the interfaces would enter into the baseline before or during design of the components that will utilize the interface. This preserves the investment in the component and increases the importance

to the contract. As discussed above, forcing the use of the formal program baseline IDD helps resolve conflicts.

### R.3.2.6 Distributed Object Systems

One final futuristic point is about distributed objects. There is much work going on in this area and it could solve a lot of the interface problems. By localizing the data and its functions into an object that is shared between applications, it localizes all the processing associated with the data into one entity (e.g., a single Ada package). The single entity can be built by a single group. This will eliminate much of the interpretation problems of today's message based systems, where the source group builds some processing, and the destination group builds the rest of the processing.

Until there are standards and environmental support for distributed objects, intelligent linkers can give some benefit today. Many Ada environments provide linkers that only include subroutines that are referenced by the program. Thus, we could create packages that encapsulate all the processing related to a message, including creation, population, transmission, receipt, usage, and release. This naturally partitions into source and destination subroutines. The source CSCI only references the source subroutines, the destination CSCI only references the destination subroutines. The linker is intelligent enough to only include the appropriate subroutines in each CSCI's image, thus optimizing image size. The package can be built by one individual or team, reducing or eliminating the possibility of ambiguity.

*In summary, manage the interfaces, and you can manage the project.*

---

## R.3.3 Integrated Product Teams

**Reduce inter-team conflicts by creating integrated product teams.**

As discussed above, team dynamics are difficult to balance. Why not avoid the balancing act as much as possible? Integrated product teams are the answer. By locating all the people responsible for building and testing a product or component under the same functional management, and ideally in the same physical location, you tear down many artificial walls. Within a CSCI, the key functional areas include systems engineering, and software test. Within a subsystem (a collection of similar or tightly coupled CSCIs), support software and CSCI-CSCI integration should be joined with the development team.

Here are some examples of the benefits of eliminating these walls. If the requirements generation and software design are tightly coupled, there will be more flexibility and support for the spiral development model. The requirements vital to CSCI architecture can be prioritized first. Complex requirements can be prototyped to see what makes sense, and what is unrealistic. Including software test on the same team allows them to be involved in more discussions, allowing them to achieve a better understanding of the CSCI under test. The manager must not allow the independent validation process to be compromised, but the benefits far outweigh the risks.

On a larger, subsystem scale, combining support software and CSCI-CSCI integration with the development teams removes several problematic walls. Support software is often a lower priority. By including them on the same team, there is more flexibility of moving people around, more sharing of knowledge, more assurance that the true requirements for support software will be built. Traditionally, one of the biggest walls is between the development team and CSCI-CSCI integration. Here is where the individual pieces of the

product must come together into a system. Here is where early requirements, interface or design flaws will show up for the first time. Tension can easily build and finger pointing is a natural response. Finger pointing can build walls in a flash. If we are all part of one big team, the possibility of everyone being committed to an integrated product increases. There are less turf wars. You should never hear a developer saying to an integrator, “I don’t have a requirement to do that.” An integrated team like this should all adopt this extra requirement — “we shall build a product that works reliably, and meets the budget and contractual obligations.” Adding this requirement eliminates a lot of finger pointing.

One specialty note on integrated product teams. Often there are organizations with specialized integrators, or integration support teams. These teams often are the wise old lab gurus. They have the ability to set up procedures and techniques that will help other integrators and developers. Often, there are also several CSCIs that form the services layer of functionality, the layer that other applications will utilize to perform their tasks. This includes the operating system, inter-program communications, display interfaces, resource management, and data management. These two groups have tightly coupled responsibilities, yet they are often chasing conflicting priorities and deadlines.

By combining the two teams, you have the ability to influence the design of the service layer with hooks and angles that will pay for themselves many times over in integration.

*In summary, combine teams for better productivity and efficiency.*

---

### R.3.4 Reuse

**Reuse offers an often elusive payback.**

Reuse, like process definition, is an investment and an attitude. It requires an investment by your best engineers to find what should be reusable, and build it so it is easy to use and reliable. This also helps with the attitude, having respected engineers sign off on a product makes its adoption easier. Sometimes, the attitude requires some legislation. Engineers are always trying to create. Trying to get them to reuse a component gives the perception of removing some of the creativity. But if the policy is established and the cost savings obvious, they will stop complaining, stop trying to show why they can’t reuse a particular component (or why they could build a better component), and get on with it.

When dealing with reuse across teams, solid cost models must be established. These models must take into account the extra cost of building something for reuse as well as the payback when something gets reused. Middle management often gets measured on cost. Who would volunteer to spend extra money to build something reusable if there wasn’t a mechanism to reflect the extra cost of construction and the payback when it is reused? How successful a reuse program is often comes down to deciding whether or not to make an investment in good people to manage and implement the program.

One overlooked possibility for reuse is a common CSCI software architecture. This helps in several ways — by using a proven architecture, each development team would not have to spend time integrating the basics — it would be a given. Integration of CSCIs would be simplified, and people would be more able to adapt to new assignments on other CSCIs. Understanding the basic CSCI architecture would need to be done once. Understanding each CSCI’s unique part would be the only task on a reassignment.

Most CSCIs in a distributed system can initialize in the same basic way, elaborate, open their communications agents, request disk-based data, signal that they are ready to run, then run. And all but the hardest real time CSCIs could utilize the same type of input / output mechanisms and tasking structure. The use of common Ada package specs, and either unique package bodies or separate subprograms gives developers a compile-time binding to the common architecture.

*In summary, reuse is an investment and an attitude.*

---

### R.3.5 Waterfall versus Spiral Model

**The waterfall model has many shortcomings for large systems.**

There is much enthusiasm in the literature for the spiral model of software development — design a little, code a little, test a little. On a project that must follow MIL-STD-2167A, and is too large to comprehend by one person, it may be difficult to decide where to start designing. Plus, the typical contract is structured with big, waterfall type events such as SSR (System Segment Review), PDR (Preliminary Design Review), and CDR (Critical Design Review). This waterfall contract encourages everything to proceed in parallel. In reality, some pieces (e.g., CSCIs) will progress more quickly than others, and some pieces are needed to mature earlier than others.

Why not structure the waterfall events in phases? Establish a layered approach to the system with the service layer first. Then add the other functionality into phases based on its criticality to the mission. Write the contract to have the SSR, PDR, and CDR for each phase. This allows the critical layers to mature when necessary. It also allows more resources to concentrate on the biggest risk areas up front. Having the service layer built first gives many benefits. It allows all future development and integration to start with a solid system foundation. It allows tools and integration techniques to be matured as development and integration needs dictate. If the service layer were being built concurrently, there is often not enough resources to add the integration hooks and handles. But if the service layer is established, mature, and baselined, the layer experts could easily enhance it to grow with integration's needs.

Ideally, CSCIs will interface only with other CSCIs from the same phase. This sounds nice but will rarely happen in practice. In this case, we suggest a phased delivery for the CSCIs. Perform some top level design on the interfaces with CSCIs in later phases, but only build the portion that interfaces with CSCIs in this or earlier phases. This may sound radical but countless dollars are spent on rework because of immature interface contracts. By the time the later CSCI really evaluated the interface, it was found to be inadequate, thus wasting all the time spent by the earlier CSCI. Now for trivial interfaces, they can be specified early, but for any nontrivial interface, it pays to wait until both sides are really ready to sign a binding contract.

In keeping with the spirit of the spiral model, the contract must grow too. But it must grow together. This lock-step approach to the spiral model has some nice benefits. A functional system is ready much sooner, and much more often. Updates have less of a ripple effect since both sides of an interface have waited to take the next step together. This scheme causes some different tracking models to show progress. Functional threads, spanning more than one CSCI and the associated messages, become the ideal tracking mechanism. It is relatively easy to identify the messages and software components that support that thread, thus it is relatively easy to track their implementation progress. A good Configuration Management (CM) system will assist in this task by allowing related changes to be grouped or tracked together.

Using the spiral model also moves emphasis from documentation to functionality. The sooner this shift occurs, the better. The documentation is essential but it should not drive the program. Integrated product teams also greatly assist with implementing the spiral model.

*In summary, structure the contract to map to the spiral model.*

---

## R.3.6 Configuration Management

**Without a solid CM process, you will struggle.**

Configuration management (CM) is important to any size project. It is absolutely required for a large project. All items generated on the project should be controlled in some manner. This includes all requirements, white papers, designs, code, test data and results, integration scenarios and results, everything. Now the degree to which each piece of information is controlled varies. Some items require customer approval to change. Milestones such as CDR trigger the transition from informal (or developmental) (DCM) to formal CM (FCM). Other items such as software development folders are always under DCM and never under FCM. One can think of the difference in the degree of formality in terms of who should approve changes or proposed changes. The less critical or smaller the chance of a ripple effect, the closer the approval should be to the person making the change (if there is no chance of a ripple effect, it may be appropriate to have no approval). The more critical or higher the chance of a ripple effect, the more people should be involved with approving the change. Ideally, the DCM and FCM environments should be one in the same. As the degree of formality increases, the approval simply increases to the appropriate level(s).

CM serves as a natural communication mechanism. Using only baselined items to perform all software builds, document generation, etc., provides a controlled mechanism to help mature the system (a nice complement to the spiral model). Once the community buys into utilizing CM properly, product integrity improves. This can be achieved for little cost. CM doesn't have to be expensive, but not having CM is very expensive. Having DCM and FCM can provide some other benefits as side effects. Some obvious benefits are automated build support. Further, automated tools can be run on the baseline to calculate metrics, and perform compliance and consistency checks. The life cycle can be modeled in the automated CM systems. This modeling allows the developmental and maintenance process to be enforced, automated, and tracked.

For example, let's assume the process for a given CSCI is design, code, test, then baseline into FCM. As the design review is completed, the librarian could enter the review results into the system and the system would automatically register the items that completed the review. This could then be used to calculate some "percentage complete." The same analogy could be used for code and test. Once baselined, the items should only change for additional functionality or rework. In either case, the change causes the items to revert back to some previous process phase (either design or code in this example). By indicating the items that must change, the process phase, and an estimate of the effort to implement the change, the system can keep track of how much additional functionality or rework is outstanding.

Some specific items that require extra attention are interfaces. These can be inter-CSCI interfaces or intra-CSCI interfaces. We've already discussed inter-CSCI interfaces and the importance of managing them, including configuring them. For intra-CSCI interfaces, utilizing Ada and the package spec, we can achieve similar control and benefits. Placing a more formal level of control on package specs elevates their importance

and reduces the risk when developing in parallel. For example, after a design review the package specs may be placed into DCM and the CSCI team leader must approve all changes to them.

*In summary, establish a complete CM process — don't start coding without it.*

---

## R.4 Technology

On a program this large, it may seem odd that we aren't talking much about technology. There were some high-tech developments in fault tolerance and sonar. But as far as the use of technology to manage the program, there wasn't too much to talk about. We had a variety of homegrown interface and integration tools that were utilized with much success. We had an enormous UNIX development and integration environment that gave us the horsepower needed to manage the vast amounts of data and code. There was always a search for silver bullet tools that would alleviate this or that problem. Most proved to be of arguable value. The only technology we want to single out is the Ada language itself.

---

### R.4.1 Use of Ada

**Ada was designed for large programs.**

We feel no other language (except possibly C++) would have survived on a program of this scale. The benefits provided to us by Ada were tremendous. Sure, we stressed the Ada environment often, and broke it more than a few times. But the benefits far outweighed the problems. You've no doubt heard about the benefits of information hiding. We've already discussed the use of Ada for interfaces. Below are some additional benefits and techniques learned.

#### R.4.1.1 There's No Substitute for Experience

Ada is a rich language. It provides many features that need to be used in moderation on a large, real-time program such as this. Experience with the run-time properties of some of Ada's more mysterious features paid off time and time again. In examples where a team did not have access to an experienced Ada person, often the results showed. Some of the dangerous features were over use of generics and tasks, use of variant records (caused many run-time problems), too much nesting of generics and packages, and too much information in the package spec (resulting in global objects).

#### R.4.1.2 Self Documenting Code

Maintenance on a large program such as this is actually performed ongoing. People leave or get reassigned and someone needs to pick up and finish where the originator left off. Reliance on external documentation often proves to be futile. When well designed, Ada code can be self documenting. Naturally, there were exceptions to the rule. But when done properly, the maintenance payoff was felt early.

### R.4.1.3 Use Tasks Wisely

Tasks are not free. In a real-time system, improper use of tasks can wreak havoc. But when used properly, tasks provide a natural expression of the real concurrency problem being solved. When possible, utilize a common CSCI architecture. Here are a few suggestions on the use of tasks.

- Utilize the main program — it is in reality a task. We suggest using the main program to orchestrate the application's major state changes (e.g., initialization, shutdown, etc.). This allows a single reference to understanding each application's state transitions. By using common package specs for the common components of each application, the main program can even be common (yielding a common architecture). Even if it isn't, using this approach provides a consistent scheme among all CSCIs.
- Utilize an event management scheme within tasks, as appropriate. An event management scheme is a method where multiple, simultaneous events are managed within one task. This allows for more concurrency without more tasks. By constructing a common event management package early, you enable designers to reduce the number of tasks needed.

### R.4.1.4 Integration Techniques

Several Ada environments provide excellent program debuggers. These tools are tremendous in isolating problems. Adding TEXT\_IO statements to display intermediate states, events, and results also help isolate many problems. However, there will undoubtedly be problems that can't be found or even isolated with debuggers due to the timing or real-time nature of the problem. TEXT\_IO may cause performance problems and therefore be impractical for all problems. We offer the following hybrid approach.

Sophisticated logic analyzers exist that can unobtrusively capture data from a system. By taking advantage of the analyzer's strengths, namely the ability to capture data from a specific address range, we can add instrumentation to key algorithms and functions and gain a tremendous insight into the underlying system. For example, we might create a block of 10 integers, each one corresponding to some key function. Whenever one of the key functions invoked, say a memory allocation routine, the function writes a code (in this case, the amount of memory allocated) to the corresponding integer. The code should be designed in such a way as to allow analysis of the operation through the logic analyzer. The analyzer simply captures all "writes" to these integers. Simple tools can decode the results. The instrumentation doesn't add any significant overhead, but the results can help find many tough problems.

### R.4.1.5 Design for Tuning Up Front

Integrating a large system will require many intermediate test environments. There won't always be the complete system to work with. Some components will need to be simulated, some will need to be turned off. Designing up front to support tuning is one of the most important factors for success. Utilizing the service layer and / or configuration files to turn on or off functionality, can provide the needed integration flexibility. With this mechanism in place, the TEXT\_IO approach to logging events and data can also be useful.

### R.4.1.6 Ada Is Not for Everything

Ada is not ideal for everything. Operating systems and low level functionality may be better served with C or assembler. Graphical User Interface functionality can benefit from automated tools that generate C or

C++ code. Accepting this fact, you next need to deal with a mixed (Ada to non-Ada) environment. As discussed in the interface section above, do not treat this environment lightly.

*In summary, Ada has what it takes for large systems.*

---

## **R.5 Summary**

Models such as that presented above should help guide medium to large scale programs. Given the proper management support and risk management strategies, the processes and technology for tackling tomorrow's complex systems exists today.

---

## **R.6 References**

- [1] Tom Peters, "Thriving on Chaos", Harper Collins.
- [2] Dr. Steven Covey, "The Seven Habits of Highly Effective People", Simon and Schuster.
- [3] F. Gregory Farnham and Kevin J. McSweeney, "Going to Sea with Ada", Defense Electronics, October 1994.
- [4] F. Gregory Farnham, "Lessons Learned on BSY-2", Software Technology Conference, April 1993.

---

## **R.7 About the Author**

Robert F. Sullivan Jr. is Vice President of Technology and Product Development at PROSOFT, Inc., located in Syracuse, NY. He is responsible for development and improvement of PROSOFT's state of the art Configuration Management product, XStream.