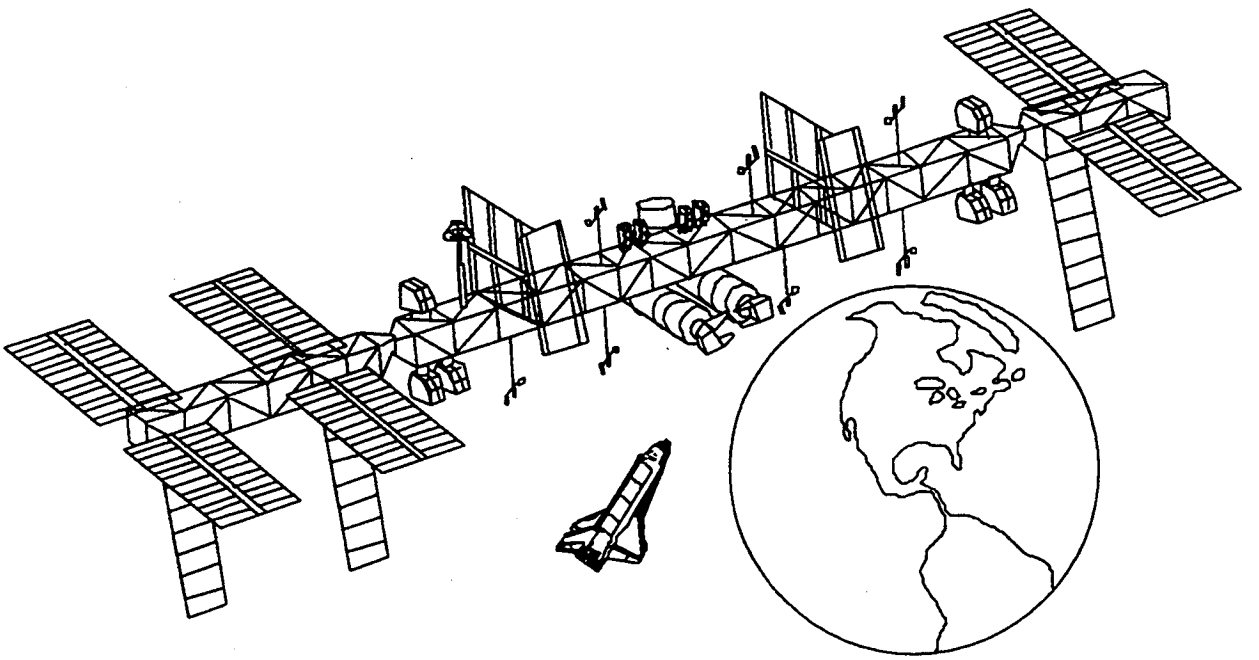
N94-11431

LARGE PROJECT EXPERIENCES
WITH
OBJECT–ORIENTED METHODS
AND
REUSE

DECEMBER, 1992



Prepared for:

Seventeenth Annual Software Engineering Workshop
Software Engineering Laboratory
Goddard Space Flight Center
Greenbelt, Maryland 20771

Prepared by:

W. Wessale, CAE–Link Corporation
D. Reifer, Reifer Consultants, Inc.
D. Weller, CAE–Link Corporation

Introduction

The Space Station Verification and Training Facility (SSVTF) is being developed by CAE–Link Corporation under contract to the Mission Operations Directorate (MOD) at the Johnson Space Center (JSC) in Houston, Texas to provide full mission training for both Space Station Freedom (SSF) crew and ground flight controllers. Figure 1 presents a block diagram of the SSVTF. The SSVTF consists

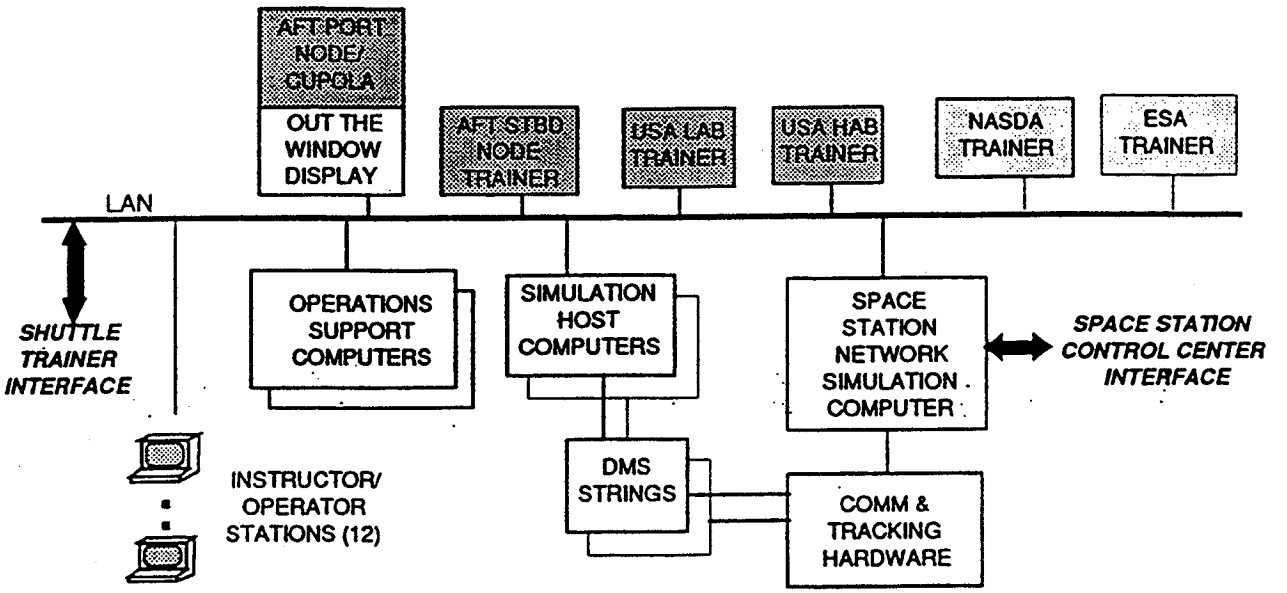


Figure 1 – SSVTF Block Diagram

of four crew stations with supporting computing and communications equipment. The four crew stations represent the SSF USA Habitation Module, USA Laboratory Module, Node, and Cupola. These crew stations can be used concurrently for simultaneous independent training; combined with each other or the Shuttle Mission Training Facility (SMTF); or integrated with the SSF Control Center. In addition, the SSVTF can be used without crew stations to drive the SSF Control Center for flight controller training.

The SSVTF will be used for assembly, shuttle proximity operations, and SSF systems operations training. The SSVTF will be used to train both astronaut crews and ground based flight controllers. This includes the use of the Space Station Remote Manipulator System (SSRMS) via closed–circuit television (CCTV) and out–the–window viewing from the cupola on top of the Node. The SSVTF will include the SSF Data Management System (DMS) — the onboard computers.

The SSVTF will also be used to develop and verify crew and flight controller operational procedures for mission planning and mission support.

The SSVTF is a large project with over 1.8 million lines of source code estimated to be delivered by 1999. There are five deliveries planned between 1995 and 1999, providing increasing capabilities corresponding to the build–up of the on–orbit vehicle and the control center. The first delivery is a Part Task Trainer (PTT) in January of 1994, the second delivery is the initial Full Task Trainer (FTT) in June of 1995. The second Delivery consists of over 500K source lines of code (SLOCs) and the largest delivery will consist of approximately 600K SLOCs of code.

Due to pressures from reduced budgets, maintaining concurrency with the actual SSF vehicle and ground systems, and a projected 30–year life span, the SSVTF project is attempting to leverage the following state–of–the–art software engineering technologies:

- Ada as the primary programming language, with C++ in limited areas
- Object–Oriented Requirements Analysis (OORA) followed by Object–Oriented Design (OOD)
- Incremental development lifecycle
- Systematic reuse
- CASE tools; both front–end analysis tools, and back–end reverse engineering / code analysis tools
- Networked Unix workstations with Rational Ada development environment

The remainder of this paper presents our experiences with each of these technologies and the lessons learned applying them.

Development Lifecycle

Strategy

Incremental Development

The incremental lifecycle being used for SSVTF software development breaks up each Delivery to NASA into multiple internal deliveries from the engineering departments to a Delivery Manager. Each of these internal deliveries is called a Capability Build Release (CBR), and represents some subset of the requirements allocated to one CSCI. CBRs were defined working backwards from the fixed Ready–for–Training (RFT) date. CBRs were defined by a Delivery Integration Team that used knowledge of the critical development path, the technical risks, GFE availability dates, inter–CSCI dependencies, and the SSF on–orbit assembly sequence. The overall goal was to begin integration early and continue integration in parallel with the remainder of the development. We wanted to maximize execution time on the target computers within a target environment to increase reliability of the delivered software.

Each CBR applies to one CSCI and represents a subset of the system level requirements that must be satisfied by the CBR. The development engineers can then subdivide each CBR into one or more Increments. Each increment is scheduled and managed separately, and proceeds through the full software engineering lifecycle independently of all other increments. Most CBRs consist of two to three increments. Increments are integrated and tested at the CSCI level to form a CBR before they are turned over to the Delivery Manager for integration into the Delivery.

Typically, the first increment develops a capability that is simple, or well understood. The first increment familiarizes the developers with the process, the products, and the tools. The second and subsequent increments are chosen to mitigate risk. That is, the second increment would implement the capability that has the greatest risk:

- Capability risk (may not be able to fully provide the capability)
- Technical approach risk (envisioned approach may not be adequate)
- Schedule dependency risk (may need to develop portions of the capability early to support the needs of another CSCI)

Inspections

Inspections were used rather than formal Reviews. The SSVTF Inspections are not formal inspections by an independent QA group, but rather engineering inspections. SSVTF Inspections are chaired by a multi–disciplinary Software Review Board (SRB) formed within the SSVTF project. The SRB consists of 7 members representing each of the development organizations and covering all development domains — real–time, simulations, graphical user interfaces, system software, software development tools, operations support tools, and information systems. These members were selected based upon their technical abilities, and the respect of their peers within their engineering domain.

Inspections are scheduled for each CSCI by the lead engineer. Attendees include one or more SRB representatives, the developer(s), the customer's cognizant engineer(s), and representatives from other organizations that plan to use or interface with the CSCI. Inspection materials are distributed prior to the inspection meeting, and attendees are expected to have reviewed the materials.

The actual inspection meeting is issue–oriented. That is, the inspection meeting is not a walk–through of the inspection materials, nor is it a summary presentation of the inspection materials. Rather, attendees raise issues identified while re-

viewing the materials, and the issues are either answered, an agreement is reached on the corrective action, or an action item is recorded for further analysis. Meeting minutes are used to record and manage the deficiencies. There are no Review Item Dispositions (RID) generated that require official review, response, disposition meetings and closure signatures.

Inspection materials are snapshots of engineering products taken from Software Development Folders (SDF). These materials represent the engineer's working notes at the time of the inspection. They are typically formatted in accordance with the deliverables, but no technical editing, quality assurance. or formal delivery occurs.

The inspections were initially created as a mid–course guidance and feedback mechanism to assist in the application of Object Oriented Requirements Analysis (OORA). Each engineer had received formal training in OORA, but since there was not an in–place OORA culture, some means was needed to provide early guidance to the engineers before they had expended too much effort or schedule, and to provide a means to identify special cases that were not covered by the OORA class materials.

Lessons Learned

Get your customer to buy–in when incremental development is used

Incremental development requires your customer's acceptance since it impacts the structure and content of your development schedules and makes it more difficult to assess overall project status. Your customer must understand the benefits provided by incremental development: shortened development schedule, greater reliability in the delivered product, and facilitating risk management by isolating risky capabilities, or attacking them early.

Inspections make more sense than PDR & CDR

Inspections can focus on the real issues — requirements, performance, and design — rather than become "dog and pony" shows or training sessions.

Logistics are much simpler. Materials can be produced and distributed by the developers rather than formally delivered through a contractual mechanism. Inspections can be scheduled as needed, facilitating incremental development. Teams can be built and focused on the issues at hand. Finally, Inspections are smaller and are limited to just those people that are either directly impacted by the CSCI or have previous applicable development experience.

Delivery Manager concept is essential

The engineering managers develop CSCI increments to satisfy the CBRs, but a Delivery Manager is necessary to define the CBRs, manage the integration of the CBRs, and manage the selloff to the customer. This allows the deliveries to stay focused to the customer's needs, while the engineering managers continue development of CBRs for future deliveries.

Beware the negative effects of prototyping

Prototyping can hinder the culture change to Ada, Object–Oriented methods, and reuse if it is done prior to formal training in the project's methods and tools. Without guidance or formal training, the prototype groups will evolve their own methods, notations, and styles. They will be reluctant to redo their work once they have received formal training. The prototype group becomes a culture that strives to preserve itself.

Prototyped software that exhibits a portion of the desired behavior will be viewed by the developers and managers as a deliverable product. After all, if the software already does what it needs to do, why not just tweak it to satisfy the rest of the requirements, instead of starting over and rebuilding it?

Processes

Strategy

Existing processes would not work with the chosen object–oriented technologies and pre–defined tools. Existing processes within CAE's NASA programs were developed to support manual, functional methods for FORTRAN deliverables We performed an internal SEI Software Engineering Capabilities Assessment (SECA) and determined that we needed to initiate an earnest effort to improve our processes, especially as we moved to object–oriented technologies and Ada. Change was needed because processes existing within CAE from DoD projects using Ada did not utilize

OORA, were tailored to DoD–STD–2167A, and assumed the use of a different development environment.

We established a Process Engineering Group (PEG) within the SSVTF Program Office to facilitate, and coordinate the creation, dissemination, and application of processes for Object–Oriented development with Ada. The PEG worked with engineers to develop prototype processes that would then be applied and refined before being published.

We also established a Process Improvement Steering Committee (PISC) to gain approval of the processes from the engineering and program managers. The PISC assured us that a process would not violate schedule or budget constraints.

Processes focused on product contents and inspections, rather than the engineering methods to develop the products. We taught methods in formal classes, but we established completion criteria in the processes. We found it necessary to define who did what, and when, to efficiently mechanize the Inspections.

A Software Review Board (SRB) was established, as described above, to oversee the application of the processes and provide guidance and interpretation to the engineers.

Lessons Learned

The most critical processes are the ones without sex appeal

The most critical processes are those that describe how different groups interact — who does what and when. Critical processes were: the development lifecycle; the process for each inspection; the process for publishing and releasing deliverables; the process for creating, screening, acting on, and closing Review Item Dispositions (RIDs). We needed product checklists and protocols to define when products were complete, who received them, who reviewed them, and how to handle reviewer's comments. These processes were especially important since they had to be revised or created from scratch to accommodate the development lifecycle object–oriented methods and reuse. Existing conventional processes would not work within an incremental development lifecycle where there is extreme parallel development.

Surprisingly, the non–critical processes were the engineering methods used during each phase of development. We used formal training and examples rather than processes to define these engineering methods. The optimum engineering process varies from domain to domain and group to group depending upon: group expertise, available data, difficulty of applying the methods, schedule constraints, and tools used.

Just–in–time process engineering works when you dedicate the resources needed

Due to schedule and resource constraints, we ended up applying just–in–time principles to establishing processes. In reality, we applied just–a–little–late principles, because the first couple CSCIs were used as prototypes to develop and verify the processes. These CSCIs suffered more rework and less–compliant products than the remainder. By the time the first CSCIs reached PDR, our process definition activities had matured so that we were finally just–in–time.

We have found that just–in–time process engineering requires a constant staff of 2–3 people in the Process Engineering Group (PEG), with support from responsible engineers designated for each process. The responsible engineer would develop the checklists, product examples, and rough process flow. This material would be applied to the first couple of CSCIs, then turned over to the PEG for formalization into a process. A process typically requires two to three months elapsed time from designation of a responsible engineer until it's released as an approved process.

Process groups can be small, and should focus on work flows, checklists, and examples

The key to success with process packaging was written procedures and checklists. We used processes to specify who does what and when through work flows, product completion checklists, and realistic examples. We used training for application of methods.

Our Process Engineering Group consists of only 2 full–time engineers with part–time assistance of designated engineers from the developer's organizations. The part–time assistance from engineers equates to roughly one more full–time person. This

approach was supported by the formal methods training as well as mentoring from the SRB.

Need a multi–disciplinary team to promote and mature processes

We created the Software Review Board (SRB) to review all products, and provide uniform guidance on the application of the methods. The SRB is composed of representatives from each of the engineering managers, and has a cross section of domain expertise. In–Process and completion inspections were established for each CSCI to provide early feedback and guidance for each phase of development. The SRB also provided guidance on real–world situations that were not covered by the formal training.

Methods

Strategy

We wanted a consistent method applied throughout the development lifecycle. We had learned from the B–2 Weapon Systems Trainer (WST) that: 1)object–oriented design was effective, 2)it was difficult to transition from functional decomposition to object–oriented design at PDR, and 3)object–oriented design created more opportunities for reuse.

We also wanted to facilitate large–grained reuse. Therefore, we wanted to apply OORA followed by OOD with Ada.

We applied an OORA method and notation that is a hybrid of many different methodologists (Grady Booch, James Rumbaugh et al, David Harel, and CAE methodologists). We adopted class specifications and composition/inheritance diagramming concepts from Booch; association concepts from Rumbaugh et al; object message diagrams from Ed Colbert and Ed Seidewitz; statecharts from David Harel. The overall OORA process was developed by CAE methodologists in conjunction with our training subcontractor.

We also tailored our deliverable documentation to support object–oriented development and reuse. After unsuccessfully tailoring DoD–STD–2167A DIDs, we developed our own document, the Object–Specification (O–Spec). The O–Spec bundles requirements, preliminary design, and detailed design of each class together. Each O–Spec contains multiple sections, where each section documents one class. This improves reusability by allowing classes to be individually reviewed as a whole, modified, and even moved among deliverable documents without disturbing the remainder of the document. The O–Spec is intended to provide a single source for all life–cycle information concerning each class.

Lessons Learned

Fusion of methods is needed due to their relative immaturity

Some methods are strong in statics – composition and inheritance. Other methods are strong in inter–object dynamics — messaging. And still other methods can be used to express intra–object dynamics — states. None of the methods uniformly support all three.

This is complicated by the limitations of the tools available, and the need to address real–time performance.

We chose to merge the techniques of eight methodologists to provide complete coverage, and then alter or restrict the methods and notations to fit the constraints of our toolset.

Scalability and the Big Picture are key success attributes for a new methodology

We previously found that excessive rigor during analysis or design can lead to paralysis as the requirements or design model matures. Inevitable changes towards the end of analysis or design require inordinate administrative efforts. Rigor during analysis and preliminary design does not scale up to large projects. Some looseness is acceptable during analysis and preliminary design. The requirements or design model does not need to have semantic closure as long as it can be interpreted and understood by human reviewers familiar with the project.

We found, however, that the object–oriented methods and products tended to focus on classes, to the detriment of the Big Picture. That is, our reviewers found they could understand the individual classes, but found it difficult to verify how instances of the classes would cooperate to satisfy the CSCIs functional requirements. This was especially true for re-

quirements allocated to more than one class. This deficiency can be solved with more effort spent developing the requirements and design model at the CSCI level.

The switch to new methods / processes is accompanied by massive infrastructure changes

We found our new object–oriented methods and reuse processes had a broad impact on the project infrastructure. We changed our initial infrastructure to accommodate: new tools; new products; new processes for reviewing, publishing, and delivering the products; new ad hoc organizational entities; and new earned value measurements. These changes required as much effort and time to realize as the original change to OORA and OOD within the engineers.

A Chief Methodologist, or Methodology Team, is required to provide guidance and mature the process

A focal point is required to guide the culture change to object–oriented technology and reuse. This focal point must have early and frequent exposure to all CSCI products to gain feedback on method and process effectiveness, provide guidance to the developers, and resolve unexpected issues. Technical issues arise that must be addressed as development progresses, and application questions must be answered by mentors in each development domain.

Tools

Strategy

The SSVTF project uses the software development toolset defined by SSF Software Support Environment (SSE). The SSE is a set of software development tools that were specified by the SSE Contractor for use by all SSF operational software development activities. The SSVTF project uses a network of 96 Solbourne S4/500 Unix workstations distributed at the developers work areas, supported by 5 Solbourne Servers and 9 Rational S400 Ada engines. PCs and Macs were also available, but were only used as administrative tools.

The following tools are available at each of the Solbourne workstations:

- Teamwork/Ada – Used to create OORA diagrams. We redefined the meaning of the icons to satisfy our OORA notations. We did not use Teamwork/Ada to create machine diagrams or for code generation.
- Interleaf Technical Publication System – Used to create O–Spec documents, as well as reports and white papers.
- Alsys Ada – Used for Ada training and prototyping.
- Oracle relational database including SQL extensions and Oracle CASE tools– Used for deliverable database applications, as well as requirements tracing, interface management, and budgets.

The Rational Ada engines provide:

- Rational Ada environment – Used for editing, compiling, and testing Ada components
- Code Management and Version Control (CMVC) – Used for engineering version control during development, and configuration management during integration.
- Remote Compilation Integrator (RCI) – Used for compiling and generating executable loads on the target computers.

NELS is being evaluated as a potential reuse repository.

Due to budget constraints, we plan to maintain a ratio of 1.4 software developers for each Solbourne workstation, and 12 Ada developers for each Rational Ada engine.

Lessons Learned

Searching for proper tooling is like hunting for the Holy Grail

You won't be able to find the perfect set of tools that will:

- Support a concise and precise graphical expression of requirements and design
- Verify completeness of requirements allocation

- Verify correctness of design
- Maintain consistency of notation across multiple developers
- Maintain consistency of interfaces among multiple developers
- Produce deliverable documentation without additional effort
- Evolve production code directly from the requirements and design products
- Trace from requirements to design to code, and back
- Provide version control for developers, and configuration management for integration and sell off

There currently are no toolsets or environments that can provide all of this, regardless of what vendors tell you. Different tools each provide a different portion of these capabilities, but trying to create an integrated toolset is not practical due to the dissimilar paradigms and information representation techniques used by each tool. To be successful, you must exploit existing tool capabilities and compensate for their weaknesses.

Benefits of CASE tools are largely overrated outside of database applications

We found an interesting paradox with front–end CASE tools. If you rigorously apply the method a front–end CASE tool was developed to support, your project will become ensnarled keeping the volumes of information correct and consistent as you approach the completion of analysis or design. On the other, hand, if you don't rigorously apply the underlying method, the front–end CASE tool is used for little more than a drawing program with rubber–banded arrows.

Rational is both good news and bad news

The Rational Ada environment supported by CMVC is a very powerful development environment that takes the sting out of strong typing, yet still accentuates all of the programming–in–the–large capabilities of Ada. The Rational Subsystem, provided by CMVC, permits incremental or parallel development, which is especially important on projects with multiple development teams. The Rational scales up well for large projects.

The Rational environment also provides the concept of "subsystems" that facilitate parallel development and minimize recompilation effects. The structure of Rational subsystems is critical to both rapid compilation and management of parallel development efforts. Rational subsystems must be defined as a part of the preliminary design activities.

The Rational environment requires a significant learning curve — several months — over and above the learning curve for the Ada programming language. This learning curve is due to:

- The Rational environment is unlike anything else.
- It introduces a completely new vocabulary.
- It provides a complex set of capabilities, is very flexible, and is extensible
- It does not adhere to the normal paradigm of code, compile, link, execute

When formal training is supplied, the Rational can be applied very effectively.

Our most valuable tools were organizational

We constantly struggled with getting the development organizations to agree on lifecycle milestones and deliverables, to agree on methods and approaches, and to consistently apply them. We were undergoing a culture change, and the organizational tools that supported that change were more important than the workstation tools used for development.

We used the following organizational tools:

- We established formal training classes for OORA, OOD, the Ada programming language, and our tools.
- A System Engineering Team was established to develop consistent interpretations of requirements.
- A Software Review Board (SRB) was established to develop consistent guidance for software methods, architecture, and programming.
- We established internal inspection milestones along the development lifecycle to

provide visibility into the application of methods and tools, and to provide early corrective guidance to the developers.

- We established a Process Improvement Steering Committee (PISC) to gain engineering and project management acceptance and approval of software development processes.
- We used brown bag presentations at lunch to share technology and information relevant to SSVTF.
- We used Unix netnews as a forum for questions, opinions, gripes, and guidance.

We did not reorganize the project. We established ad hoc boards and teams to provide the additional organizational support.

Reuse

Strategy

The SSVTF budget has been based on systematic reuse since 1989. We used the software estimating tool SoftCost–Ada (Reifer Consultants Inc.) to account for the effects of reuse in the SSVTF development budget. Productivity during the first deliveries was penalized by 15% to account for the extra effort required to systematically identify and construct reusable components. Productivity during later deliveries was increased by 25% to account for exploiting the reusable components.

Ad hoc reuse occurs when developers independently discover and exploit reuse opportunities during design and coding of software components. Systematic reuse occurs when it is first identified prior to detailed design, responsible developers and reusers are identified, and the resulting agreements are reflected in development plans. Systematic reuse is required to achieve large–grained or widespread reuse.

We formed a Reuse Inspection Team and identified a Reuse Inspection milestone during OORA to facilitate systematic reuse. The Reuse Inspection Team consisted of seven software engineers representing each of the engineering managers and each of the development domains. The Reuse Inspection Team chaired a Reuse Inspection for each CSCI and used preliminary OORA products to identify reuse opportunities. Each opportunity was documented by a Reuse Action Item (RAI) that identified the affected developers, and requested that they evaluate and recommend a reuse approach. Acceptable recommendations included: 1)agreeing that two or more identified components be acquired by reusing another component without change; or 2)merging requirements to specify a larger, more complete component; or 3)recommending that separate components be developed without reuse.

A separate reuse group responsible for developing and maintaining all reusable components was not considered practical. The established SSVTF organization was organized around product areas such that many reusable components would span two or more engineering managers. We were also concerned that a separate reuse group would lose its focus on project requirements and become less responsive to the schedule dependencies of the individual reusers. Engineering managers would be reluctant to risk their development schedules. We felt this would eventually undermine reuse. We wanted reuse to occur through cooperative efforts among the engineering managers so they had an active role in making it happen.

Although we are considering using an incentive system in the future, we have not used an incentive system to date. We have approached reuse as a cultural issue — it's something you're expected to do.

We anticipate additional reuse impacts managing change of reusable components during integration.

Lessons Learned

Barriers to reuse are primarily managerial

We found that most developers are willing to identify, plan, and follow–through on systematic reuse. We also found that upper management likes the purported cost and schedule benefits. The difficulty lies in schedule dependencies, budget impacts, and trust among the middle–managers. These technical leads and engineering managers must make reuse happen, and bear the burden if it doesn't.

Normally the designated developer is the user with the most complete or complex requirement for the reusable component. However, a dilemma occurs when the developer's need date follows that of other potential reusers. Can the developer expend the resources in time to meet the reusers need dates? If

not, can one of the other users (who have less complete or complex requirements) successfully develop the more complete reusable component, and without impacting their schedule?

If the reusable component is larger in scope than originally planned by the developer (and it usually is), can budget be made available from the reusers to augment the developer's budget?

In addition to schedule and budget issues, how can a reuser be sure that what the developer is creating will really be reusable and still satisfy his requirements once it's done? Will the reusable version of a component still satisfy real-time performance constraints?

We solved the schedule and budget problems through work sharing at the developer level. That is, a team of developers from two or more reusers would be informally created to develop the reusable component. This team would coordinate their activities so that each of their schedule need dates is satisfied within the original budget allocations. No formal paper work was required. These teams were "gentlemen's agreements" between the developers and their leads.

We addressed the issue of trust by having the reusers participate in the reusable component's inspections.

Strategic planning, commitment, empowerment, and persistence are needed to make reuse happen

Reuse intentions will wither and die before they bear fruit if not tended. Developers and managers lose sight of reuse as they attack their immediate technical, schedule, and budget problems. Original reuse agreements can be compromised, and new reuse opportunities can be overlooked, if reuse isn't revisited repeatedly during the development lifecycle.

We have made reuse a part of every inspection — either as a major topic, or as a check that previous agreements have not been compromised. This provides six opportunities to readdress reuse during development.

We empowered the developers to analyze each reuse opportunity and recommend a reuse approach. We made initial reuse decisions based on developer's recommendations. If the developers recommended against reuse, and their technical leads supported that recommendation, no reuse would be planned. On the other hand, when reuse was recommended, the potential reusers were always able, with their technical lead's assistance and support, to identify a developer and an approach to overcome schedule and budget impacts.

Object-oriented methods make it easier to spot, assess, and agree upon reuse opportunities

Composition and inheritance diagrams made it easy to spot reuse opportunities. A comparison of attributes and operations made it easy to spot differences in required components. Reuse analyses often resulted in alternate or additional class definitions.

We held 35 reuse inspections that generated 53 reuse action items which resulted in 23 reuse opportunities. Based on our design at PDR, we predict that approximately 30% of our delivered software will be reusable.

We were also able to facilitate the common use of COTS products across three different development groups.

Be prepared to modify your processes to accommodate reuse

New processes must be developed to initiate, track, and manage systematic reuse, such as: Reuse Inspections, Reuse Inspection Teams, Reuse Action Items.

Milestones will require more peer involvement across development groups. That is, potential reusers as well as technical management and the customer must be involved at inspections.

Changes to reused components require approval from more than just the developer. A control board containing a representative from each reuser must approve reusable component modifications to prevent degrading their reusability.

Training

Strategy

We used three types of formal training supplemented by brown bag lunch presentations and open help sessions to support the culture change required. The three types of formal training are:

- Large Object–Oriented System Engineering (LOOSE) training for all engineers during subsystem design
- OORA, OOD, and Ada programming training for all software developers
- O–Spec familiarization training for all customer personnel and managers that would be reviewing our deliverables.

LOOSE training was created internally and consisted of a two–day class in object–oriented thinking for subsystem partitioning and the selection of CSCIs. LOOSE focused on objects, messages, and composition, but did not introduce classes or inheritance.

We acquired a single OORA, OOD, and Ada programming training vendor, Fastrak Training, via a full–and–open competition. We wanted a single vendor for consistent terminology, uniform progression from requirements to code, and tailoring for SSVTF tools, target environments, and products. Training consists of a four–day OORA class, a five–day OOD with Ada class, and twelve days of Ada programming training provided in two classes. As of the 4th quarter of Fiscal Year 1992:

- Over 160 contractor and NASA personnel have been trained in OORA
- Over 60 contractor and NASA personnel have been trained in OOD with Ada
- Over 200 contractor and NASA personnel have been trained in the Ada programming language

The O–Spec familiarization training was internally created and consists of a two–day class in reading, analyzing, and verifying the OORA and OOD products delivered in the Object Specifications (O–Specs) at PDR.

Lessons Learned

Use a single training vendor

The methods and products should flow from one step to another, and the terminology should be consistent. The analysis and design methods should build on one another without redundancy or overlap. There should not be a concept shift between analysis and design. This can only be achieved from a single vendor.

Plan to spend 3–4 staff months tailoring off–the–shelf courseware

You will not be able to buy "training" off–the–shelf. You will only be able to buy education. That is, you want your engineers to return to their desks after completing the training class and begin to apply what they were taught. Off–the–shelf courses, however, will be too generic. Engineers will return after completing an off–the–shelf course, and will immediately become confused and frustrated trying to figure out how they can apply what they were taught.

Off–the–shelf courseware must be tailored to your project before it can be effective training. Tailoring is required for:

- Project constraints: tools, architecture, COTS, GFE, existing software legacies
- Project lifecycle: phases, milestones, products
- Project vocabulary
- Project domain: examples and exercises

We incorporated domain specific examples and sample products into our courseware.

Plan on each O–O course — OORA or OOD — requiring two to three staff months of tailoring. The Ada programming course should require very little tailoring.

Customer training is essential prior to reviews and deliveries

Management and customer training is essential to facilitate an effective review. Reviewers and managers don't need to perform OORA or OOD, but they must be able to read OORA and OOD products, understand them, and verify that they satisfy requirements.

A minimum of two–days training is required to avoid wasting the reviewer's and your time. Without training, the inspection meeting will be spent describing the meaning of the material rather than discussing requirement and design issues. An ineffective review may give the appearance that no substantive issues exist, but it is more likely that an ineffective review has overlooked problems that will be more difficult and expensive to remove later.

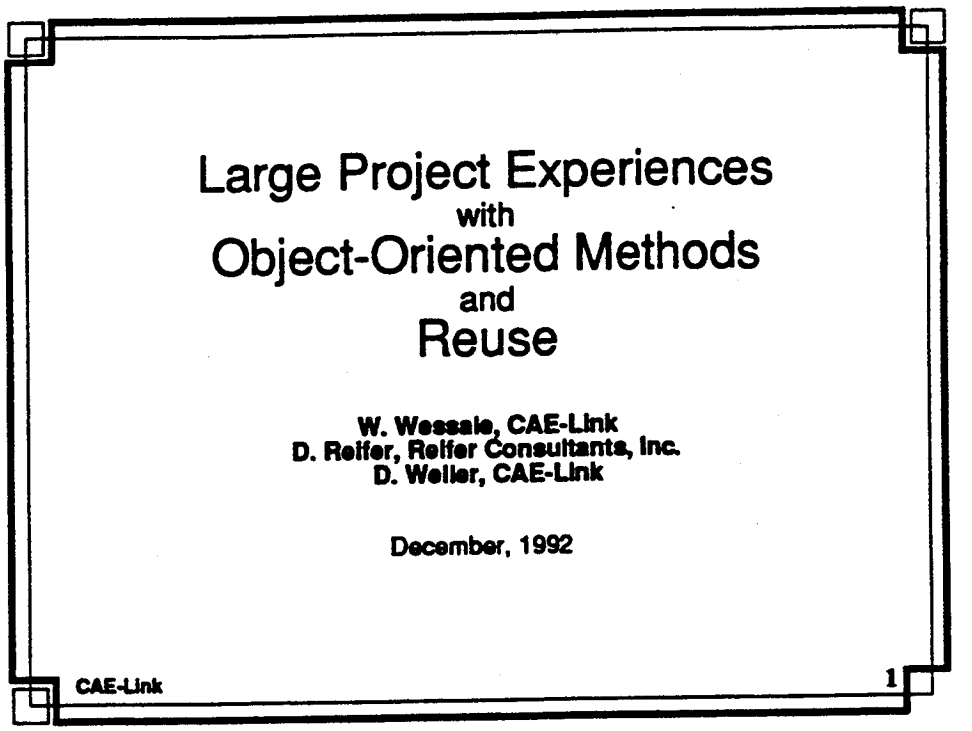
Realistic examples are key to gaining acceptance to new methods and deliverables

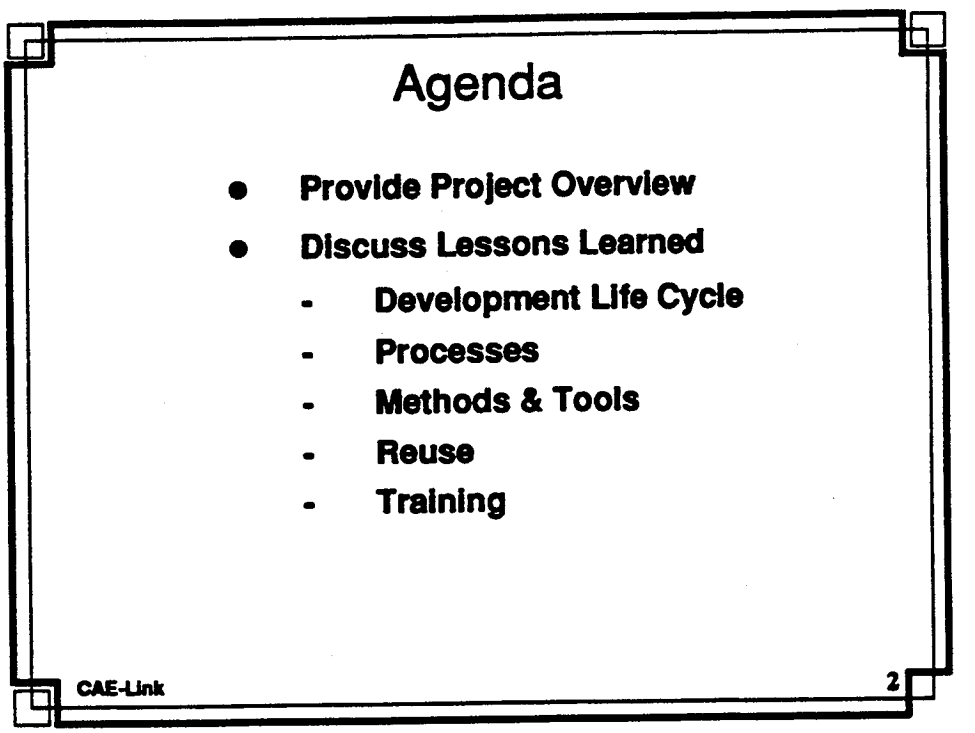
Examples help bridge the gap between the classroom and the desired project results. Examples layout level of detail and completeness expectations. Realistic examples must be relevant to the project's problem domain, and must be representative in size, complexity, and abstraction.

We developed two examples for SSVTF — an example of a real-time simulation, and an example of an on–line data processing application. We developed a vertical slice though each example. That is, we performed a complete preliminary requirements analysis to identify the objects and classes; then we completed the detailed analysis for only a portion of the classes. We specified a preliminary design for each of the classes that were fully analyzed, but we only fully implemented a portion of the operations in each class.

Summary

The SSVTF project is completing the Preliminary Design Review of a large software development using object–oriented methods and systematic reuse. An incremental development lifecycle was tailored to provide early feedback and guidance on methods and products, with repeated attention to reuse. Object oriented methods were formally taught and supported by realistic examples. Reuse was readily accepted and planned by the developers. Schedule and budget issues were handled by agreements and work sharing arranged by the developers.


Large Project Experiences
with
Object-Oriented Methods
and
Reuse
W. Wessale, CAE-Link
D. Reifer, Reifer Consultants, Inc.
D. Weller, CAE-Link
December, 1992
CAE-Link
1

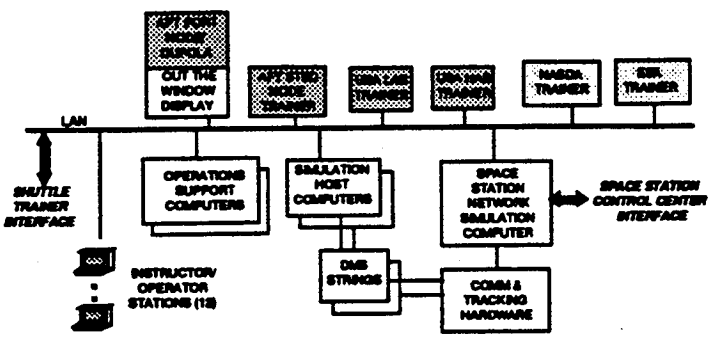

Agenda
Provide Project Overview
Discuss Lessons Learned
Development Life Cycle
Processes
Methods & Tools
Reuse
Training
CAE-Link
2

Project Overview

- **Space Station Verification & Training Facility (SSVTF)**
- **Completely new SSF training facility**
  - **SSF crew & flight controller training**
  - **Hab, Lab, Node, Cupola trainers**
  - **Assembly, Shuttle proximity operations, SSF systems operations**
  - **Procedure Verification**

CAE-Link 3

SSVTF Block Diagram



CAE-Link 4

Project Overview (Contd)

- Fundamental Issues
  - Large software development
    - 1.8MSLOCs total by 1999
    - 1.4MSLOC real-time & on-line software
  - Five deliveries
    - 600KSLOCs in largest
  - Repeating budget pressures
  - Concurrency with SSF
  - 30 year life span

CAE-Link 5

Project Overview (Contd)

- Adopted state-of-the-art S/W technologies
  - Ada and C++
  - Object-Oriented Analysis & Design
  - Incremental development life-cycle
  - Systematic reuse, not ad hoc
  - CASE tools (front-end and back-end)
  - Networked workstations w/Rational servers

CAE-Link 6

Development Lifecycle

- Incremental Lifecycle for each of the 5 SSVTF Deliveries
  - Multiple internal releases from each CSCI
    - Capability Build Release (CBR) to test & integration
  - 1st increment is simple
    - Learn methods, tools, process
  - Follow-on increments reduce risk
    - Capability, technical approach, schedule dependency
- Prototyping used for risk reduction

CAE-Link 7

Development Lifecycle (Contd)

- Inspections, not formal Reviews
  - Used new milestones for reuse, math models, user interfaces, and architecture
  - Scheduled on a CSCI basis
  - Working group, issue-based
    - Not summary presentation of deliverables
  - Issues, agreements and actions in Minutes
    - Not RIDs
  - Review snapshots from Software Development Folders (SDF)
    - Not formal delivery of entire documents

CAE-Link 8

Development Lifecycle Lessons Learned

1- Beware the negative effects of prototyping
   - Can hinder culture change
2- Get your customer to buy in when incremental development is used
   - Harder to manage
3- Inspections make more sense than PDR & CDR
   - Focus on real issues
   - Provide early feedback to developers and methods
4- Delivery manager concept is essential

CAE-Link 9

Processes

- Existing processes would not work with adopted lifecycle and technologies
- O-O Methods and tools judged to be relatively immature
- Established a Process Engineering Group (PEG) within Program Office
- Developed prototype processes based on theory, goals, pragmatics
- Focused on product contents and inspection process
- Established Software Review Board (SRB) to oversee application & maturation of the methods & processes

CAE-Link 10

Processes
Lessons Learned

1- **The most critical processes are those without sex appeal**
   - **Inspections, CM, DM, deliveries**

2- **Just-in-time process engineering works when you dedicate the resources needed**

3- **Process groups can be small, and should focus on work flows, checklists, and examples**
   - **Support methods training**

4- **Need a multi-disciplinary team to promote and mature processes**

CAE-Link 11

Methods

- **Emphasis on consistent method throughout development**
  - **No concept or structure clashes**
- **Object-Oriented Requirements Analysis (OORA)**
  - **Hybrid of: Seidewitz, Booch '91, Harel**
- **Object-Oriented Design (OOD)**
  - **Based on Booch and Simulation Virtual Machine (SVM) architecture**

CAE-Link 12

Methods Lessons Learned

1- Fusion of methods is needed due to their relative immaturity
   - Ignore whining from purists

2- Scalability and developing the Big Picture are key success attributes for a new methodology

3- Switch to new methods/processes is accompanied by massive infrastructure changes

4- A Chief Methodologist, or Methodology Team is needed to provide guidance, interpretations and mature the process

CAE-Link 13

Tools

- SSF Software Support Environment (SSE)
- Solbourne SPARC Unix workstations
  - Interleaf
  - Teamwork/Ada
  - Alsys Ada
- Rational Ada engines
  - Code Management & Version Control (CMVC)
  - Remote Compilation Integrator (RCI)
- NELS being evaluated for reuse repository

CAE-Link 14

Tools
Lessons Learned

1- Searching for proper tooling is like the hunt for the holy Grail

2- Benefits of CASE tools are largely overrated outside of database applications

3- Rational is both good news and bad news

4- Our most valuable tools were organizational

- Aimed at group consensus and team support

CAE-Link 15

Reuse

- Budget based on Reuse
- Insufficient schedule and scope too broad for domain analysis
- Strategic reuse emphasized
  - Reuse Inspection Team formed to identify opportunities
  - Reuse action items tracked
- Separate reuse group not considered reasonable
- Looking at incentive system

CAE-Link 16

Reuse
Lessons Learned

1- Barriers to reuse are primarily managerial

2- Strategic planning, commitment, empowerment and persistence are needed to make reuse happen

3- Object-Oriented methods make it easier to spot, assess, and agree upon reuse opportunities

4- Be prepared to modify your processes to accommodate reuse

CAE-Link 17

Training

- Added O-O thinking to systems engineering activities to aid in selecting CSCIs
- Acquired a single training vendor via full-and-open competition
  - OORA (4 days)
  - OOD w/Ada (5 days)
  - Ada Programming (12 days)
- Conducted 2-day training on read, analyze, and verify the products distributed at inspections and PDR
- Supplemented formal training with brown bag and open help sessions

CAE-Link 18

Training Lessons Learned

1- **Use a single training vendor for all your courses**

2- **Plan to spend 3-4 staff-months of effort tailoring "off-the-shelf" courseware to your methodologies and environment**

3- **Customer training is essential prior to reviews and deliveries**

4- **Realistic examples is key to getting acceptance from the troops**

CAE-Link 19

Summary

- **Described the SSVTF project**
- **Presented our lessons learned as of PDR**
  - **Development Life Cycle**
  - **Processes**
  - **Methods & Tools**
  - **Reuse**
  - **Training**

Keys to success:
- **Defined architecture**
- **Defined methods, formally taught, supported by examples**
- **Defined processes**
- **Feedback & guidance, early and often**

CAE-Link 20