6146'14

Software Development and Test Methodology for a Distributed Ground System

Presented by:
George Ritter
Marshall Space Flight Center
Lockheed Martin Space Operations
256-544-8269
george.ritter@msfc.nasa.gov

Technical Contributor:
Pat Guillebeau

Abstract

The Marshall Space Flight Center's (MSFC) Payload Operations Center (POC) ground system has evolved over a period of about 10 years. During this time the software processes have migrated from more traditional to more contemporary development processes in an effort to minimize unnecessary overhead while maximizing process benefits. The Software processes that have evolved still emphasize requirements capture, software configuration management, design documenting, and making sure the products that have been developed are accountable to initial requirements.

This paper will give an overview of how the Software Processes have evolved, highlighting the positives as well as the negatives. In addition, we will mention the COTS tools that have been integrated into the processes and how the COTS have provided value to the project.

EHS Software Design Methodology
Table of Contents

The Enhanced HOSC System (EHS)

Marshall Spaceflight Center's Payload Operations Integration Center (POIC) is home to EHS. EHS is a command and control telemetry processing ground system made up of computers, networks, and software that has been developed and put into operations in support of the International Space Station payload operations. EHS began over 10 years ago as an upgrade replacement system for the "POC 2" VAX-VMS that support Space-Lab payload operations on-board the Space Shuttle. EHS is also on-line as the ground control system for the Chandra X-Ray observatory in Cambridge, Mass.

EHS Software Development

EHS Software Development has historically followed the waterfall approach in which high level requirements known as Level A's are first developed in response to project needs. These level A requirements are decomposed into smaller subsystem requirements known as Level B's. The Level B's are used by software developers to design and code the software system. In this fashion, the system design "falls like water" from high level (very low detailed) requirements, to very low level (high level of detail) code. A complete software development process includes the testing phases in which the software must operate successfully before the system can be certified as flight ready. Software creation and software change is controlled through our software Configuration Management (CM) processes. One of the most important parts of our software CM is the way we "tag" software files that are being delivered, based on the change that drives the delivery. We tag software with HOSC Problem Reports (HPRs) and with Engineering Change Requests (ECRs).

Later in our systems maturity we have experimented in some areas with a modification to our waterfall model with a methodology called Spiral Development. Spiral Development has proven useful for areas where we are migrating existing requirements to new technologies.

Whatever form of the development process we use, our system requirements and design must be documented. The EHS design processes has been captured in a Process Definition Documents (PDD). There have been times we have diverged from the PDD, but overall it is our standard for doing business and deviations from the process require an individual waiver.

Level A Development

The Software Development process is shown in Figure 1. The process begins with Engineering Change Requests (ECRs) that drive the creation and updates of Level A requirement. The EHS has generic Level A's and project specific Level A requirements since EHS is used for multiple flight projects including Chandra X-Ray Observatory and the International Space Station (ISS). The Level A requirements are typically created by the Systems Engineering portion of our project organization, and are grouped into functional subsystems. The subsystems include Telemetry Acquisition, Telemetry Processing, Commanding, System Level Services, Web Services, Database, or Payload Information Management. The Level A's are reviewed in depth at this point by software

development to determine the feasibility, level of effort, and proposed schedule for their implementation. In the early stages of this project, Level A's were reviewed at a typical Software Requirements Review (SRR) meeting. Now that our system is in a much more mature state, these requirements, along with the assessment of the developers, are reviewed at an internal board meeting called the HMCG. The HMCG includes representatives of all POIC stakeholders. POIC stakeholders are FD41 (POIC NASA sponsor), CADRE (POIC Operations), and the contractor's (UMS) Operations Team, System Engineering, and development groups, plus representatives from remote sites that will be using EHS for their science mission aboard ISS. The purpose of the HMCG is to determine if the Project should proceed with the implementation of the requirements.

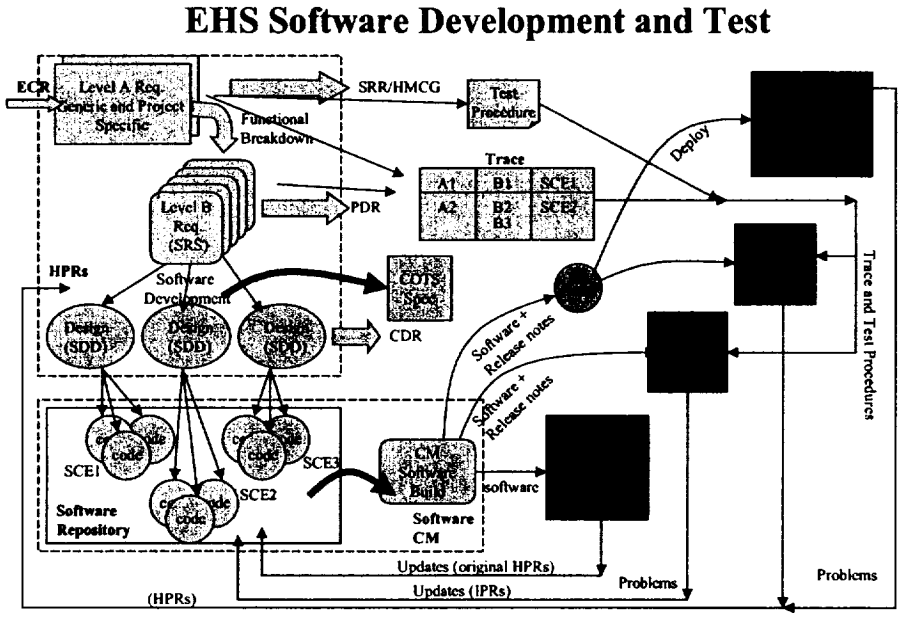


Figure 1: Software Development Process

Level B Development

Once the Level A's have been approved by the HMCG and base-lined, the requirements are functionally decomposed into smaller pieces known as Computer Software Configuration Items (CSCIs). Simply put, CSCIs are a group of requirements for which software can be created by approximately 3 or 4 individuals. Level B requirements are then developed for each CSCI and placed into one or more Software Requirements Specifications (SRSs). The level of detail for SRSs is inconsistent at best but typically must be at a level that allows you to start thinking of implementation. Each SRS Requirement is placed into a trace document along with the associated Level A requirement for which the Level B can be traced. A sample from our trace is shown in Table 1. Each Level B must be traced to insure we are not inappropriately making up

new requirements, and to insure there are no un-implemented level A requirements. This process is subjective and again requires the review and consent of all state holders.

027v1 Par	SRS 027 V1 Requirement	B Bld	Lev A Doc	Lev A Par	Level A Requirement
3.2.2.1.a	The Interface Display Operation User process shall adhere to MSFC-STD-1956.	2 | 3 | 4 | 4.1 e1.0	MSFC- RQMT-1440	14.2.1. A	The Display Operation UI shall adhere to MSFC-STD-1956.
3.2.2.1.aa	The Interface Display Operation User process shall provide the capability to enable limit/expected state sensing for all objects on a display.	2 | 3 | 4 | 4.1 | e2.0	MSFC- RQMT-1440	14.2.1. L	The Display Operation UI shall provide the capability to toggle limit sensing on and off.
3.2.2.1.ab	The Interface Display Operation User process shall provide the capability to zoom or un-zoom a time or XY plot on a display.	2 | 3 | 4 | 4.1 e2.0	MSFC- RQMT-1440	14.2.1. C	The Display Operation UI shall provide the capability to view data as updated.

Table 1: Sample Trace

The Level B requirements are officially reviewed at a Preliminary Design Review (PDR). In recent years the SRR and PDR have been combined into one review (we call it PDR) so that level A's, level B's, and traces are reviewed at once. Depending on the extent of the changes, the PDR may involve formal presentations, or it may just involve a "paper" review. Review Item Discrepancies (RIDs) can be assessed against the Level B requirements to document any issues the stakeholders might have.

Some CSCIs specify the use of COTs products. All EHS COTS are captured in the COTS Specification which is typically created at the same time as software SRSs A list of the COTS products used in the EHS Design is included in Appendix A.

Software Design

Once the Level B requirements are approved, the software development group begins the software design phase. In reality the software design phase begins during requirements specification. Prototypes are done early on for critical and high-risk areas. A more formal look at prototype development will be discussed in the Spiral Development section of this paper. In the classic approach, software developers take the Level B Requirements and covert them to software design. Our design is documented in Software Design Documents (SDDs). SDDs contain functional data flows, descriptions of the interfaces to other CSCIs, software architecture drawings, and screen dumps for designs that have a user interface. As EHS has matured, the software architecture drawing is the primary piece of the SDD that has continued to be developed. All other details are captured in code. Figure 2 is an example of a software architecture drawing for part of EHS that is developed on the xxx/Unix systems.

Architecture Diagram

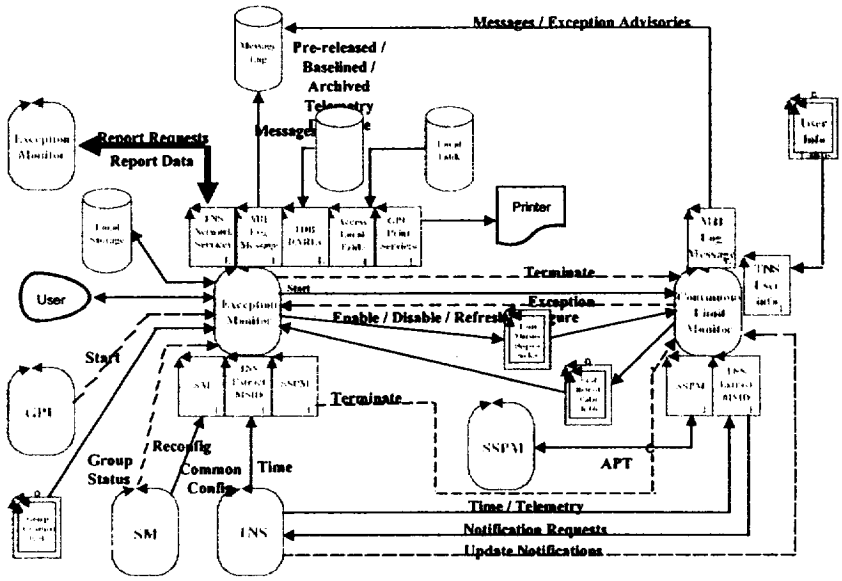


Figure 2: Sample Architecture Diagram

SDDs are reviewed at a CDR when RIDs can again be written against the design. UMS uses a Design Approval Board to determine that design is ready for coding. RIDs must be resolved and documents submitted for official updates. A Design Approval Checklist is used to insure all the bases are covered. Again, coding has probably already begun, but some of the issues raised by the CDR RIDS may remain unresolved until they are closed via the Design Review Board. This may last well into the coding phase.

Another event that takes place at design time is the creation of Software Configuration Elements (SCEs). SCEs define a particular executable or library unit. SCEs are groups of files for which an individual make-file exists in the software build. *[Please refer to Appendix B for a definition of terms used.]* SCE names are used for tracking changes in software configuration management process.

Software Coding and Unit Test

The actual creation of software is the magical part of the process. The best we can do is give software developers well understood requirements, a few important coding rules, the proper development tools, and best test environment to work through this phase. In EHS we developed a naming standard, and coding standard that is verified by running a Software Standards Verification tool (C source only). We were also required to be as "POSIX compliant as possible" in case some time in the future we needed to migrate to another POSIX compliant operating system.

One of the most important tools is the software configuration management tool. The CM tool maintains the software repository. The CM tool allows us to keep track of software

baselines and manage software change by keeping track of the reason each software source file has changed. The CM tool also allows developers to maintain different levels of sharing during the development phase. In this manner, interface code can be shared even while the next version of an interface is in work. We use Clear-case for our CM tool. Clear-case maintains a database of the software source files in what is called a Version Object Base (vob). Developers use Clear-case "views" to created and build their portion of the project. The "branch" structure of ClearCase allows developers to integrate with each other more smoothly when they are ready to move beyond unit testing.

The goal of all unit testing is 100% code statement coverage. When developers have successfully completed unit testing and basic integration testing, the source files are promoted, in Clear-case, to the CM build level. For a more detailed description of how CM manages software changes, refer to the section titled "Software CM and the CM Build Process"

Integration Testing and Code Iterations

Early testing begins at the code unit level during the code phase of development. There is some integration testing done at the unit level to avoid writing too many stubs or drivers. However, our official integration testing is broken into three phases outside of unit testing. These phases are the Development Test Environment (DTE) phase, the Development Integration Test (DIT) phase, and the HOSC Integration Test (HIT) phase. Each successive phase invokes stricter software change control then the previous phase. Iterating, or changing the code during integration test phases is crucial to progressing to more in depth levels of testing. The Level of test-team independence also increases from one test phase to the next with the HIT phase being the most independent.

The Development Test Environment

The CM build is first delivered to the Development Test Environment (DTE). The DTE hardware is separate from development hardware and should be as much like flight hardware and networks as is possible. This environment is intended to provide developers a place to verify that the CM version of their application "make" comes out working the same as the version they have been working with in their environment. The DTE is also where the next level of integration testing occurs. Applications developers drive this testing phase since the code to this point is very new and more prone to interface errors. Changes are iterated through the CM process whenever a developer chooses, as long as the code change is consistent with the approved content of the build. Code changes at this level are managed in Clear-case with the same "tag" as the original delivery. This process is shown in Figure 1 with the flow titled "*Updates (original HPRs/ECRs).*" A changed source file has it's newer version re-tagged and merged again to the CM branch for inclusion in the CM build. In the DTE phase of testing CM typically builds and loads the software on a daily basis.

The Development Integration Test Phase

The Development Integration Test Phase is primarily intended to insure that the new capabilities delivered in the current build, successfully integrate into the entire system.

In this phase we move the CM build out of the developer's area and into an area used by non-developer-testers. These test personnel work independent of the developers to verify that the software delivered meets the intent of the HPR fix or ECR. Testers have developed functional procedures for core capabilities that encompass groups of Level A requirements. In the DIT phase a subset of the test procedures are run to ensure that the overall system is intact, and any new requirements are met. Sometimes Level B requirements are used to determine what and how to test a more complex capability. For the DIT phase, developers provide preliminary release notes documenting software changes and how testing was done by developers. At the end of the DIT phase testers produce metrics that look like Table 2.

Functional Tests	Level A's	Success	Fail	HPRs Delivered	HPRs Passed	IPRs Gen'd
Time Tagged Commanding	12	12	0	1	1	0
Command Groups	15	15	0	1	1	0
Command via Scripting	30	30	0	0	0	0
Update command	12	11	1	2	2	1
Remote Commanding	25	25	0	1	1	1
Total	**121**	**120**	**1**	**8**	**8**	**4**

Table 2: DIT Test Statistics

You can see in Figure 1 that code iterations are performed in the DIT phase by identifying problems with Internal Problem Reports (IPRs). IPRs are stored in a File-maker Pro data base and developers can use the IPR as a Clear-case merge tag. IPRs are not reviewed by any stakeholders other than developers and testers. Code iterations in the DIT phase happen a couple times per week.

Once the DIT Testers are satisfied that the intent of the HPRs and ECRs is met, and that any IPRs have been corrected, we hold a Build Ready Review (BR) to officially "promote" the software to the next level of testing. At BR, CM creates a Compact Disk from their final software build. Developers turn in final copies of release notes and HPR resolutions. The CD and this paperwork make up the BR package and a copy gets stored in the project CM vault.

The HOSC Integration Test Phase

The HOSC Integration Test Phase exists primarily to verify the contents of the CD by running more system level tests on the build delivery. The CD must be good since this is our product used to load for flight. The same test procedures that were used in the DIT phase are also used in the HIT phase, but the there is more concentration on all aspects of

the functional tests and not just on fixes and updates. In addition, load testing is performed during the HIT phases. Problems found during HITT are documented with HPRs and fixes for these HPRs are can only be included n a new release of the system. At the end of the HIT phase a composite set of metrics from DIT and HIT testing is created by the test team that communicates testing statistics similar to Table 2, but with more categories such as number of HPRs and types of HPRs (regression, new-capability) generated in each phase of test. The completion of these metrics provides a method for continuous improvement by allowing analysis to be done at each phase so that test procedures can be improved based on problem reports written after the completed phase.

Software CM and the CM Build Process

The software CM group only accepts software for approved ECRs and HPRs for scheduled releases. Notice in Figure 3 how each group creates ECRs, HPRs and IPRs that flow into CM. CM takes approved change and creates a Code Baseline Checklist (CBC) for each ECR, HPR, and IPR. The developer uses that CBC to submit a merge request for the files that need to be modified or created. Software CM uses the files listed on the CBC to merge to the CM branch and create the CM Build.

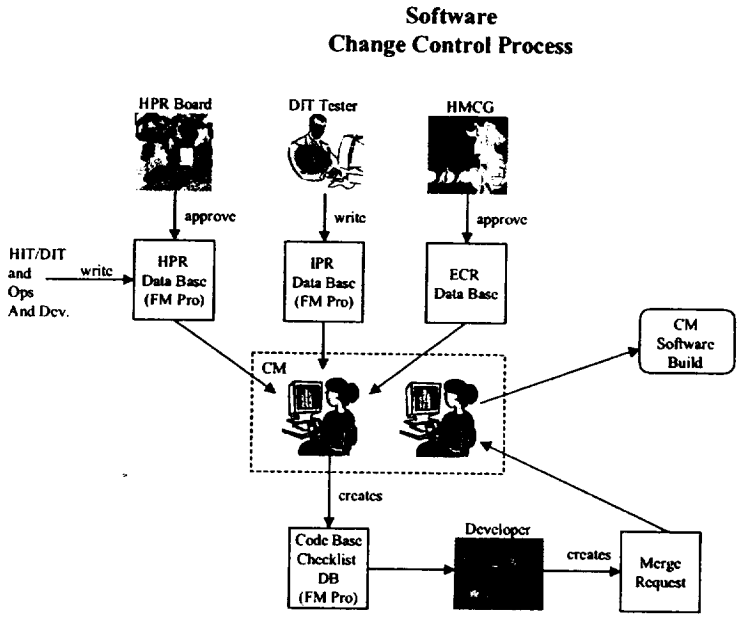


Figure 3: Software Change Control Process

Figure 4 shows more detail in the area of managing software source files. Software developers make changes to source files and then they apply a label that has been created by CM. This label is the CBC number which consists of the combination of the build number and the ECR, HPR, or IPR number (label example: HPRD2345-6.1) The CBC keeps track of the source files that have been merged for that label and it also has information such as the software developer's name, the Software Configuration Element (SCE), and the list of files merged. Software developers use the CBC form to submit

merge requests to CM. CM takes the filenames listed on the merge request, and runs a Clearcase script that finds the filenames with the labels on the developer branch, and merges the files to the CM branch. The label that was on the version of the file on the developer branch is also transferred to the new version created on the CM branch. Once CM has merged source files from the developer branches to the CM controlled branch, they initiate the software build. In this fashion, our software CM processes are automated and well understood by all contributors.

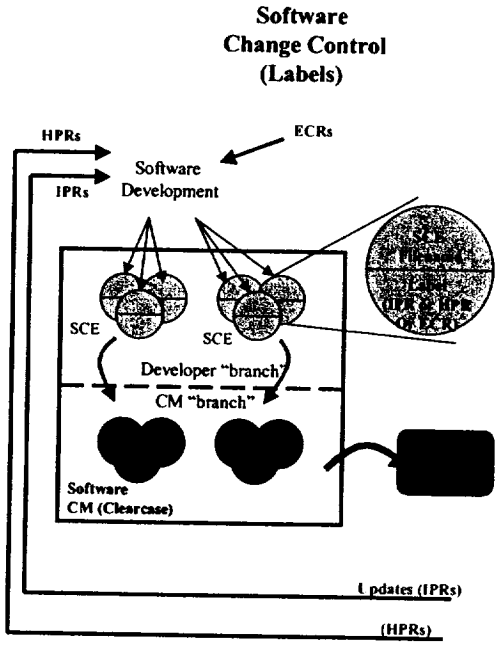


Figure 4: CM Source File Control

Spiral Development Approach

A few years ago UMS began the process of determining how to migrate our systems away from the more expensive SGI systems that have a defined end-of-life, onto commodity-based PC systems. While migrating the technologies, the base system requirements have not changed much. For this reason we needed to pick a development approach that matches the problem space.

Spiral development methodology defines a cyclic approach for growing a systems degree of definition and implementation. The Spiral Development Process must defined a set of anchor points or milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Our current system's end-of-life and high COTS maintenance costs forced us to develop a HOSC migration strategy where we were fairly confident we wanted to move the user

desktop applications to a Windows 2000 environment. This allows EHS to be run from a commodity based platform and to have EHS applications more easily integrate with the thousands of PC applications on the market. Another direction to move in is away from xxx/Unix on our server platform functions to a PC-Linux environment. Again the motivation is hardware costs and long term interoperability with future COTS products.

Referencing Figure 1, we chose existing requirements in the trace that needed to be migrated to one of the new platforms (Win 2K or Linux), designed, coded, delivered, tested, operated, re-evaluated, chose more requirements and iterated again. We updated the trace in Figure 1 to add the Build in which the requirement is satisfied on the Win 2K platform (**e2.0** = EPC version 2.0)

Another area where an iterative or Spiral development approach is best suited is in the Payload Information Management (PIMS) are. PIMS features needed to be experienced by the users for them to determine the detail requirements. Successive PIMS deliveries resulted in user's needs being extremely close to the PIMS capabilities.

Software Design Process Modifications with PC Development

While working on the HOSC migration process, we decided to try to minimize the impacts that were incurred when creating SDDs. Although the majority of the information in our existing SDDs is useful, there did sometimes tend to be too much emphasis on interface details before code was actually happening. This emphasis is inappropriate given the level of detail at SDD creation time. So we chose to represent the PC design with the architecture drawing. The architecture drawings are very useful for showing interfaces without having the code level details. Architecture drawings are also good at showing processes, libraries, and over-all design structure. Documenting the high level interfaces allows for a system wide review of the overall design.

SDDs also include a section with User Interface screen dumps that we decided could be seen when running the prototype application rather than seeing the screens on paper. So with the changes proposed to the SDD content, we decided to not create SDDs but to include the architecture drawings as part of the software CM system, and to call these drawings the Software Development Folder. When the software has to change due to HPRs and ECRs, architecture drawings are updated, then tagged and merged just like source code files.

Lessons Learned

The first lesson is don't over analyze requirements. Early on in EHS development, we spent a little too much time making perfect SRSs, presentations etc, when we should probably have been prototyping more pieces of the system. Our attention to detail at the requirements level and SDD level was very high and that costs time and also made those documents higher maintenance. When the coding began, designs changed and that invalidated a lot of documents so the documents had to be redone, and this wasted time.

The second lesson is an iterative development process with heavy user involvement is best. When we started small, and let users experience the capabilities, we found that we

developed software that more closely met user's needs. When you spend less time analyzing requirements and design documents (lesson one) you are free to prototype and work with the users.

The third lesson is don't focus incessantly on rapping up document details. Sometimes we spend days and weeks trying to close the loop on old RIDs and document updates that have an extremely low value overall. The time spent on things like this could be better spent focusing on overall product reliability and workability.

The fourth lesson is you need a test environment that MATCHES your flight environment. Time should be spent in the software design phase trying to design flight-like environments. The effort will save lots of software rework.

The fifth lesson is remain open to process improvements particularly moving from one phase of development to another. An example is our method for getting code merged. For the initial build, all new code was approved for delivery. Initially Code Baseline Checklist (CBC) requests were handwritten paper submittals used for merge requests. This evolved into a Filemaker Pro database where developers could enter information about files that need to be modified or created , print a hardcopy and submit the hardcopy to software configuration management. This evolved into our current process of having developers enter merge information into the CBC Filemaker Pro database and submit the merge request from within the database. The current merge request process is easier for those involved and less error prone.

Conclusions

The job of developing systems is a job of change. Our development processes exist so that we can effectively manage this change. If the processes are too stringent, overall productivity is slow, and we will never deliver a system in a timely fashion. If the processes are too flexible, then the change is out of control and the system quality is affected, and productivity again decreases.

In EHS development we have evolved our processes to be a combination of traditional and new. We handle our requirements in a more traditional fashion. This facilitates good test procedures and combined with the proper test environments, allows us to ensure accuracy and quality in our systems development. We have more recently migrated the design processes to a more iterative, user focused process that relies less on the document controls and more on "proof on concept" methods. In summary, we have been able to "change" the way we change, so as to develop cost effective, high quality systems.

Appendix A: Metrics; CM Builds, Lines of Code, COTs Products

This section is provided to give some perspective as to the size of EHS, and the number of different platforms and technologies that are managed.

CM Builds, Platform Types, and COTS Products

Patform/OS: Build	Platform Type	COTs
xxx/Unix: EHS, CERT, WEED, EDG	EHS Servers: Telemetry, Command, Data Base, SMAC, Login, ERIS	xxx platform "C Stuff", Failsafe, Framemaker, Networker , Java runtime/plug-in/JDK, Oracle, SQL, Teleuse, Netscape
xxx/Unix: PIMS	PIMS Server	Vfind, Draper Labs Timeliner Compiler, Java Mail, Java XML
xxx/Unix Web	Web Server	Netscape communicator/directory server/Iplanet, Java runtime/plug-in/JDK, Networker, Perl, Visibroker
xxx/Unix PDSS	PDSS Servers	
PC/Win 2k: EPC, MPS	User Client Workstation	Acrobat, Internet Explorer, Netscape Nav., Norton, MS Office, Oracle Client, X-Thinpro

Types of Code and Lines of Code that make up EHS:

Type of Code	LOC Count
.c	1.8 M
.d (Teleuse GUI code)	250K
.pcd (Teleuse GUI Definitions)	350K
Java	260K
.cc	18K
.pc	216K
.sql	152K
4GL	1.3M
.h	184K
Scripts	34K
Total	4.564M

COTs used in the Software Development Process

Product	Purpose
FileMaker Pro	HPR, IPR, CBC Data Bases
ClearCase	Source Code CM
Visio, MS Powerpoint	Software Architecture Drawings
MS Word, Software Through Pictures	SRS, SDDs,

Appendix B: Terminology

make: the process of compiling and linking software source files
make-file: the file that defines the criteria for a particular make
build: the process of running the make and creating binary images in the form of executables and libraries that are combined into a release package that is loadable.
load: the process of taking a successful Build and installing it to run on a computer
merge: the process of moving a source file from one are of control called a "branch" to another area or branch. When a source file is merged, a new version of the file is created in the CM tool on the branch where the file is merged to.
branch: and area of control. Developers have branches, groups of developers have branches, and the Software CM group has branches.
version: an instantiation of a source file

EHS Software Development Methodology

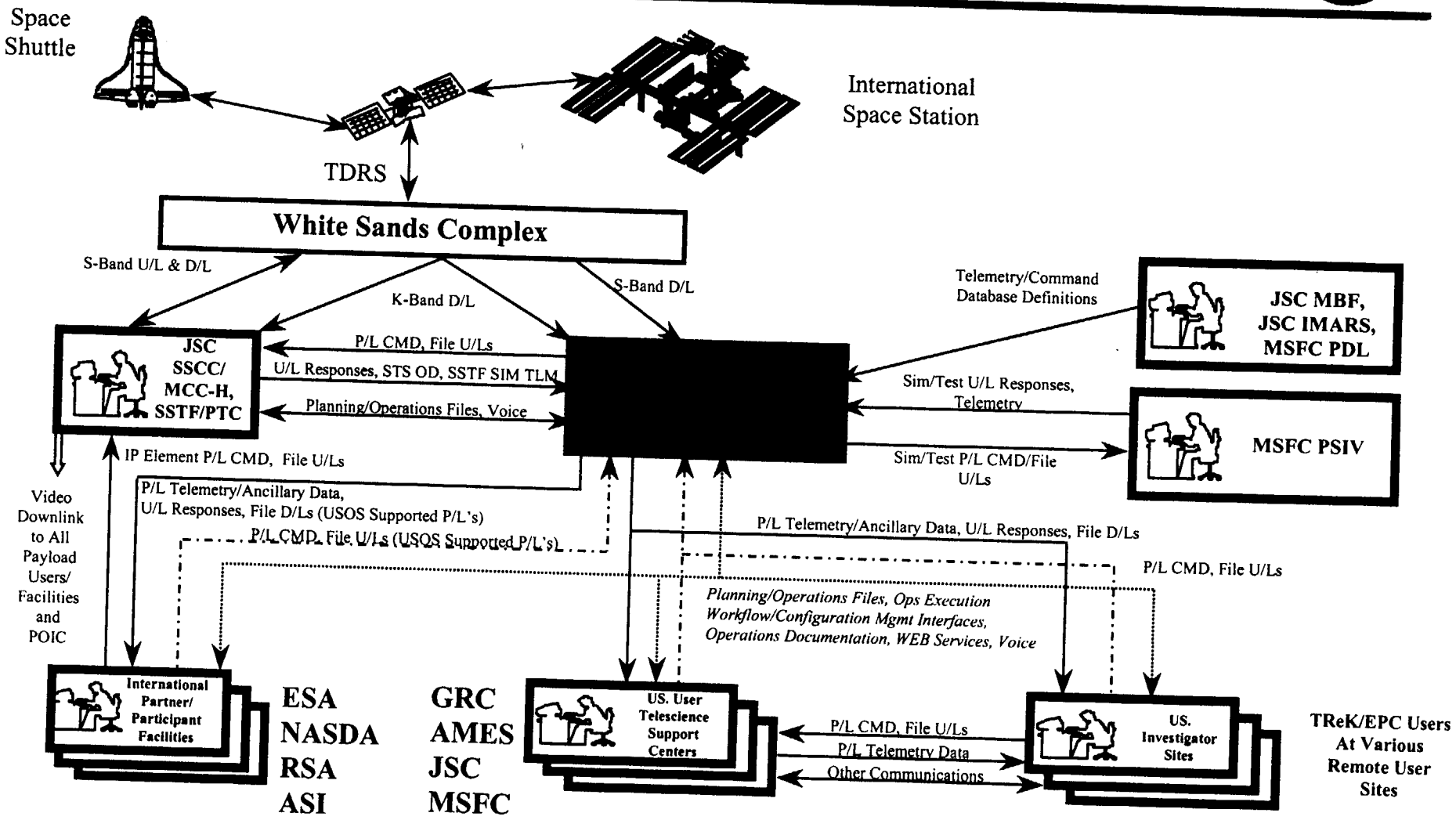
George Ritter

Marshall Space Flight Center
Lockheed Martin Space Operations
256-544-8269
george.ritter@msfc.nasa.gov

LOCKHEED MARTIN

Table of Contents

POIC/PDSS/USOC Overall Relationship To Other Space Station Program Facilities/Systems



Introduction

- The Marshall Space Flight Center's Payload Operations Integration Center (POIC) contains a command and control ground system called EHS, meaning Enhanced HOSC System
  - 'E' = Enhanced
  - 'H' = HOSC
    - HOSC = Huntsville Operation Support Center
  - 'S' = System
- EHS is the system (software, hardware, networks) that supports the ISS CADRE team in their payload support functions
- EHS is also used as the command and control ground system located in Cambridge Mass. for the Chandra X-Ray Observatory
- EHS's software development and test processes have evolved and have improved dramatically to achieve better system reliability and tighter adherence to delivery schedule.

System Size

- Types of Code and Lines of Code in EHS

Type of Code	LOC Count
.c	1.8M
.d (Teleuse GUI code)	250K
.pcd (Teleuse GUI Definitions)	350K
Java	260K
.cc	18K
.pc	216K
.sql	152K
4GL	1.3M
.h	184K
Scripts	34K
Total	4.6M

EHS COTS

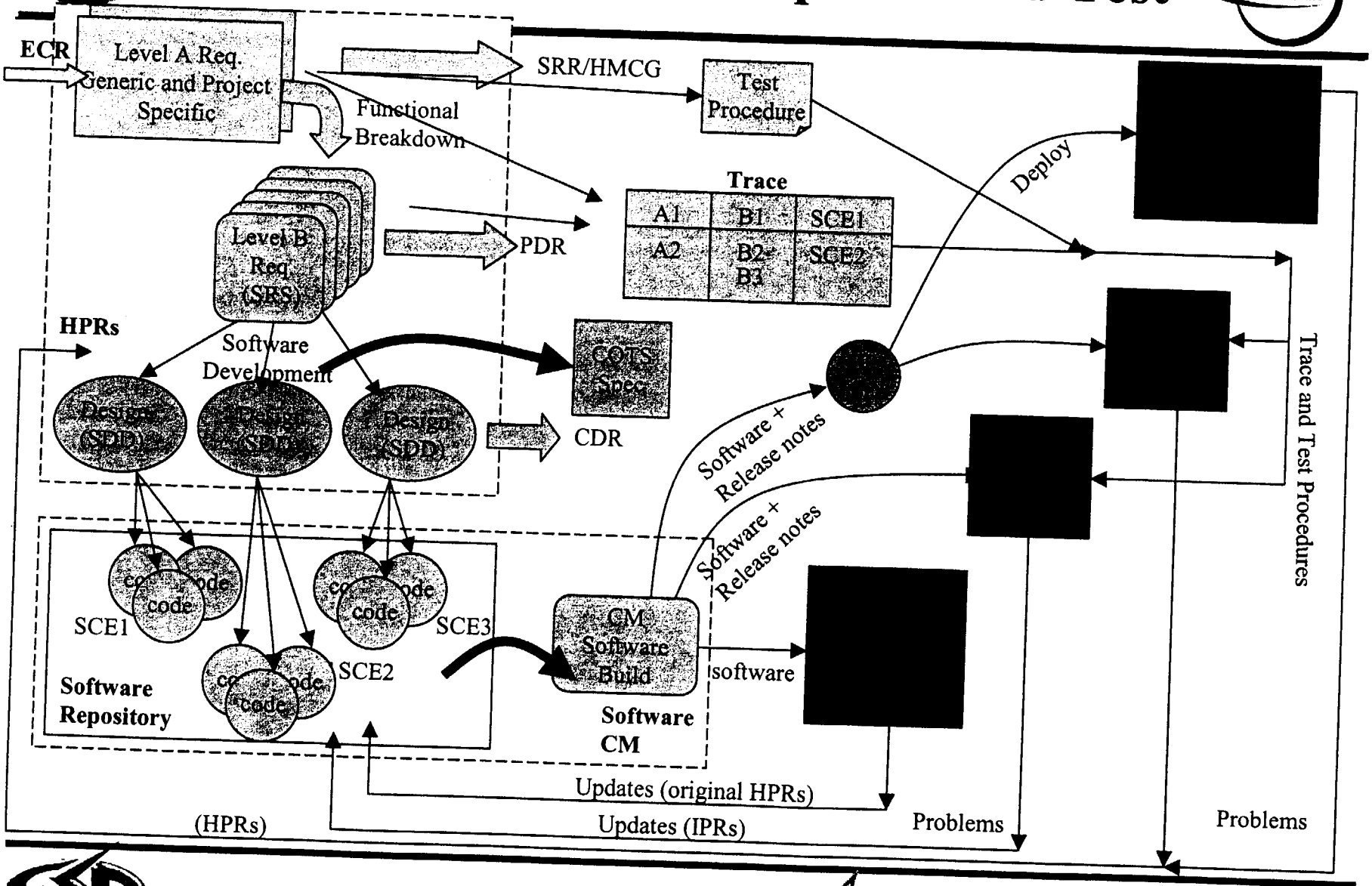
- CM Builds, Platform Types, COTS Products

Patform/OS: Build	Platform Type	COTs
xxx/Unix: EHS, CERT, WEED, EDG	EHS Servers: Telemetry, Command, Data Base, SMAC, Login, ERIS	SGI "C Stuff", Failsafe, Framemaker, Networker , Java runtime/plug-in/JDK, Oracle, SQL, Teleuse, Netscape
yyy/Unix: PIMS	PIMS Server	
yyy/Unix Web	Web Server	Netscape communicator/dir server/Iplanet, Java runtime/plug-in/JDK, Networker, Perl, Visibroker
yyy/Unix PDSS	PDSS Servers	Vfind, Draper Labs Timeliner Compiler, Java Mail, Java XML
PC/Win 2k: EPC, MPS	User Client Workstation	Acrobat, Internet Explorer, Netscape Nav., Norton, MS Office, Oracle Client, X-Thinpro

- Development Process COTS

Product	Purpose
FileMaker Pro	HPR, IPR, CBC Data Bases
Clearcase	Source Code CM
Visio, MS Powerpoint	Software Architecture Drawings
MS Word	SRS, SDDs,

EHS Software Development and Test



Level A Development

- EHS Software Development has typically followed the "Waterfall" methodology
  - ECRs -> Level A's -> Level B's -> Design -> Code
  - Level A's for each project
    - Chandra X-Ray Observatory
    - International Space Station
  - Level A's are broken into subsystems or areas
  - ECRs submit Level A updates.
  - Software Development (and Hardware Engineers) review the proposed changes to determine feasibility, level of effort, and a proposed schedule of implementation
  - HMCG Approves or disapproves changes
  - HMCG consists of the following stake-holders: NASA FD41, CADRE, Contractor System Engineering, Software Dev-Test

Level B Development

- Level A's are further decomposed into more detail Level B's.
- Level B's are grouped into CSCIs, size of about a 4 person task.
- Level B's are placed into Software Requirement Specs (SRS)
- Each Level B are then "traced" to Each Level A.
- Sample Trace:

027v1 Par	SRS 027 V1 Requirement	B Bld	Lev A Doc	Lev A Par	Level A Requirement
3.2.2.1.a	The Interface Display Operation User process shall adhere to MSFC-STD-1956.	2 | 3 | 4 | 4.1 **e1.0**	MSFC-RQMT-1440	14.2.1.A	The Display Operation UI shall adhere to MSFC-STD-1956.
3.2.2.1.aa	The Interface Display Operation User process shall provide the capability to enable limit/expected state sensing for all objects on a display.	2 | 3 | 4 | 4.1 | **e2.0**	MSFC-RQMT-1440	14.2.1.L	The Display Operation UI shall provide the capability to toggle limit sensing on and off.
3.2.2.1.ab	The Interface Display Operation User process shall provide the capability to zoom or un-zoom a time or XY plot on a display.	2 | 3 | 4 | 4.1 **e2.0**	MSFC-RQMT-1440	14.2.1.C	The Display Operation UI shall provide the capability to view data as updated.

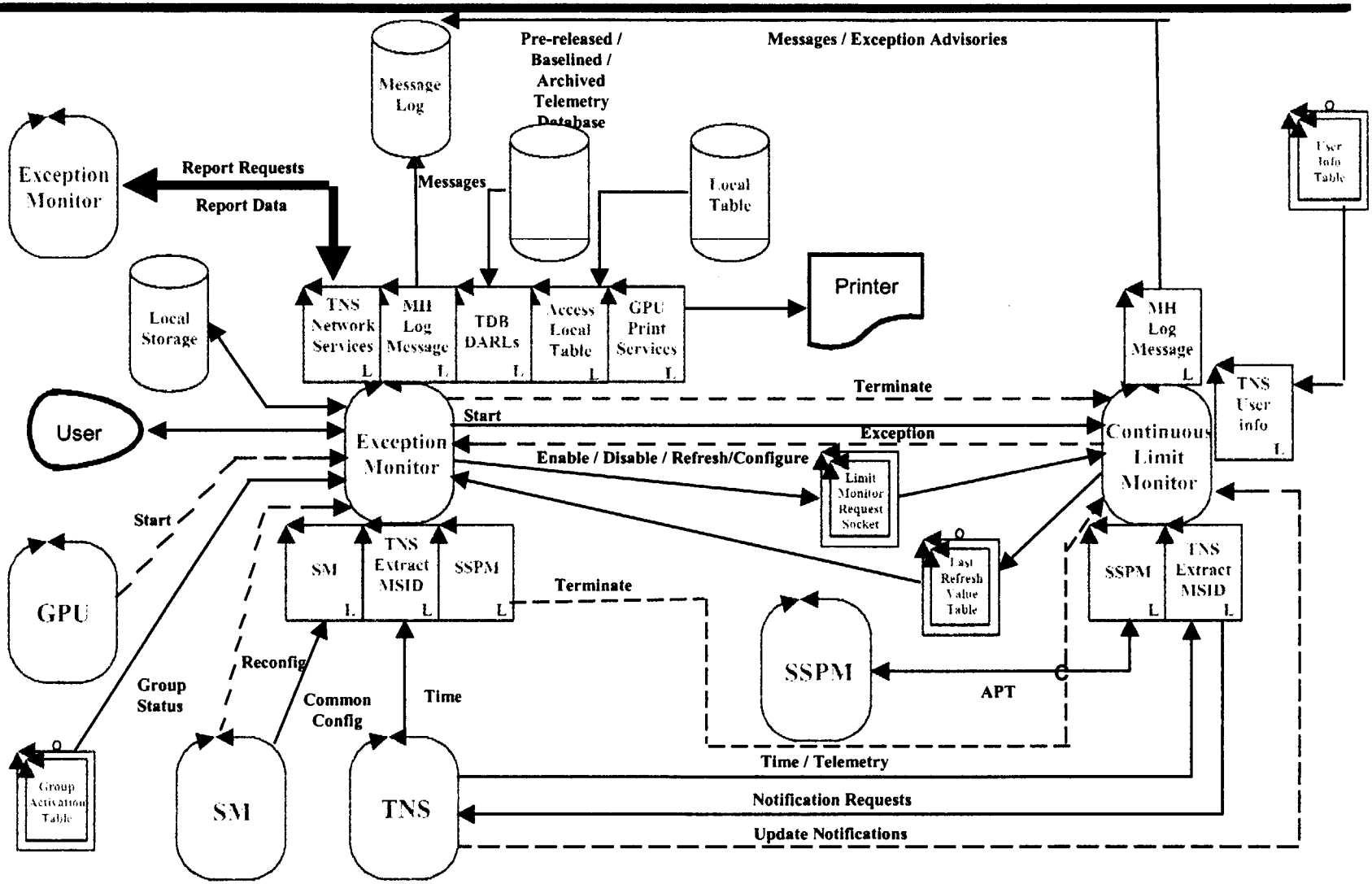
Level B Development

- Level B's are reviewed at a Preliminary Design Review (PDR)
- In recent years we have combined the SRR and PDR and call it PDR.
- Review Item Discrepancies (RIDs) can be assessed against Level A's and B's and must be resolved in order to proceed with delivery of the product.

Software Design

- Theoretically software design begins after Level A approval. In reality, coding began with the first thought of the requirement.
  - This practice has evolved over the years to allow for early prototyping
- In software Design we create Software Design Documents (SDDs)
- SDDs contain functional data flows, interface descriptions, and software architecture drawings.
- Over time we have determined the architecture drawing to be the most important part of the SDD. All other details are best found in the code it self.

Sample Architecture Drawing



Software Design

- Software design approval is required to begin coding, although in recent years we usually begin coding early and tie up design approval loose ends during coding.
  - We have a design approval checklist
  - At design approval time all Level A RIDs, Level B RIDS, and design RIDs must be in order as well as all doc updates completed
- Software design also occurs for S/W changes related to HOSC Problem Reports (HPRs).
- We also create Software Configuration Elements (SCEs) at design time.
- SCEs are the software deliverables which is typically an executable or a library or an object.
- SCEs are are group of source code files

Software Coding

- Coding is where the magic happens…. ☺
- Developers need well understood requirements, proper tools, and a proper test environment to make it happen
- We only make code changes for approved ECRs and HPRs
- One of the most important tools is the CM tool for managing source code. Our CM tool allows us to
  - manage code baselines and change to the baselines
  - Share source at various levels during development
- We use Clearcase
- Clearcase allows us our CM group to have source files on their "branch" and developers get their approved changes merged to the CM branch when a delivery is due.
  - A little more explanation for this process later on.

Software Testing

- The first phase of test happens during the code phase and is called Unit Testing.
- Developers test "units" of code in their environment and they strive for 100% statement coverage.
- Integration Testing is broken in two three phases
  - Development Test Environment (DTE) testing
  - Development Integration Test (DIT) testing
  - HOSC integration Test (HIT) testing
- Each phase invokes sticker software change mechanisms (see figure)
- Each phase moves to a unique and more highly controlled hardware environment

DTE Testing

- DTE testing is performed on a string of hardware that is in the developers area, but that is configured just like a flight system.
- DTE testing is also performed with a build of software that came from the CM group's build.
- Developers drive DTE testing but they now are using more of each other's code and they are using the CM version of everything.
- This testing is fairly general and very non-independent
- Code is iterated, or re-merged by using the original "tags" that drove the change: ECR, or HPR.
- Code iterations are frequent, CM builds and re-loads are typically nightly

DIT Testing

- The Development Integration Test (DIT) Phase is to insure that the new capabilities delivered in the current build, successfully integrate into the entire system.
- The CM build is moved out of the developer area hardware and into an area controlled by test personnel for testing independence
- Testers have developed functional test procedures that encompass groups of Level A requirements.
- Test procedures are run with the primary focus being the verification of the fixes and new capabilities (HPRs, ECRs)
- DIT Testers write Internal Problem Reports (IPRs) for problems they find related to the HPRs and ECRs that were delivered.
- Code is iterated by development about twice a week during the DIT phase based on IPR fixes.

DIT Test Statistics

- At the End of the DIT phase, metrics are reviewed

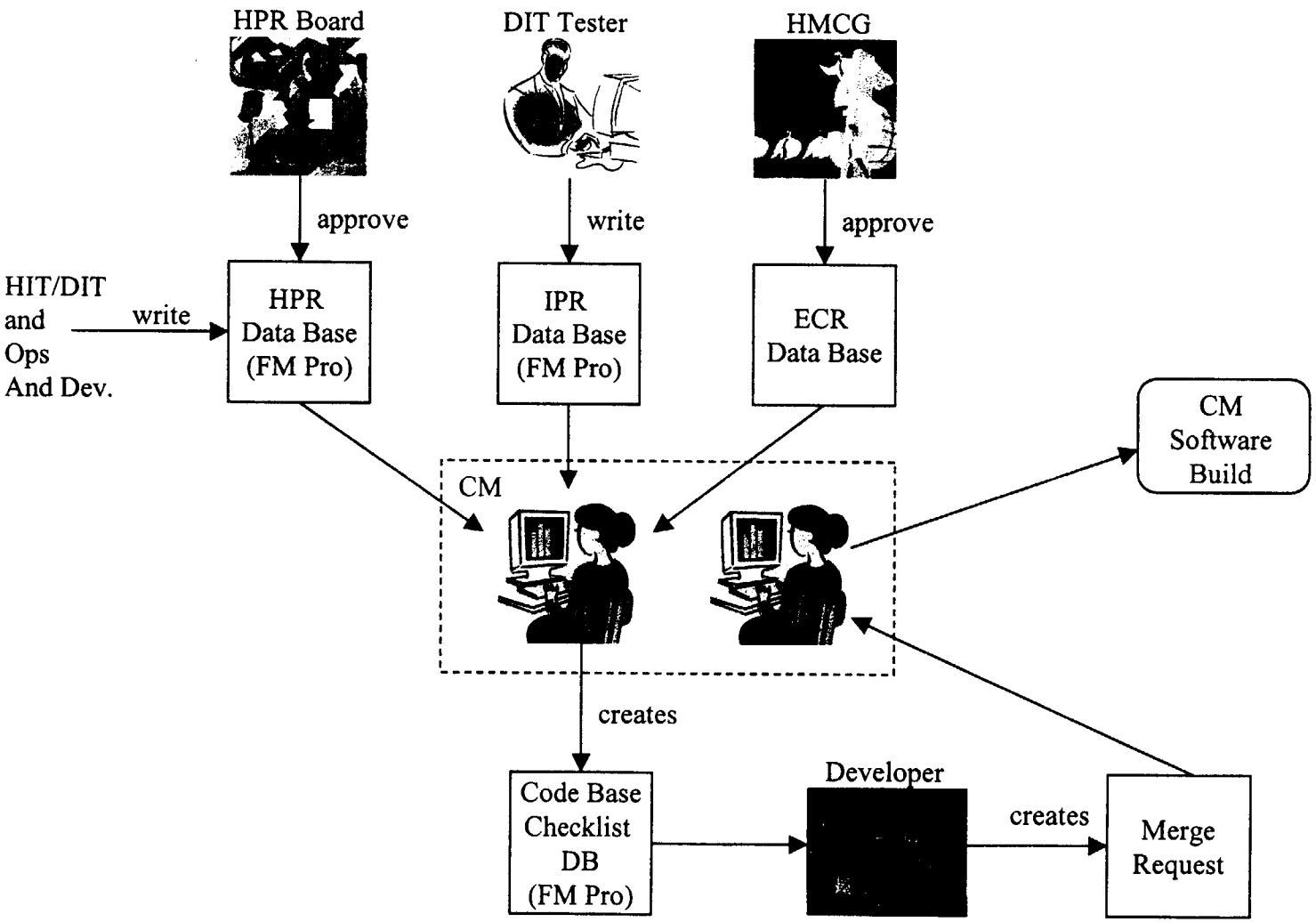
Functional Tests	Level A's	Success	Fail	HPRs Delivered	HPRs Passed	IPRs Gen'd
Commands and Data Sets	27	27	0	3	3	2
Time Tagged Commanding	12	12	0	1	1	0
Command Groups	15	15	0	1	1	0
Command via Scripting	30	30	0	0	0	0
Update command	12	11	1	2	2	1
Remote Commanding	25	25	0	1	1	1
Total	**121**	**120**	**1**	**8**	**8**	**4**

- DIT Testing culminates with a Build Ready (BR) Review at which time all paper work (release notes, HPRs) is turned in and a CD is burned with the CM build of the software package

HIT Testing

- HIT Testing exists primarily
  - to verify that the CD that was burned with the deliverable software is good
  - and to run some more "system level" tests that go beyond just fixes
  - To run load testing
- The same test procedures are used for HIT that are used for DIT, except with a wider focus.
- At the end of the HIT phase a composite set of metrics is produced that is similar to the table on the previous slide that includes information such as number and types of HPRs generated during each phase.
  - This info can be fed back for test procedure updates and continuous process inprovements

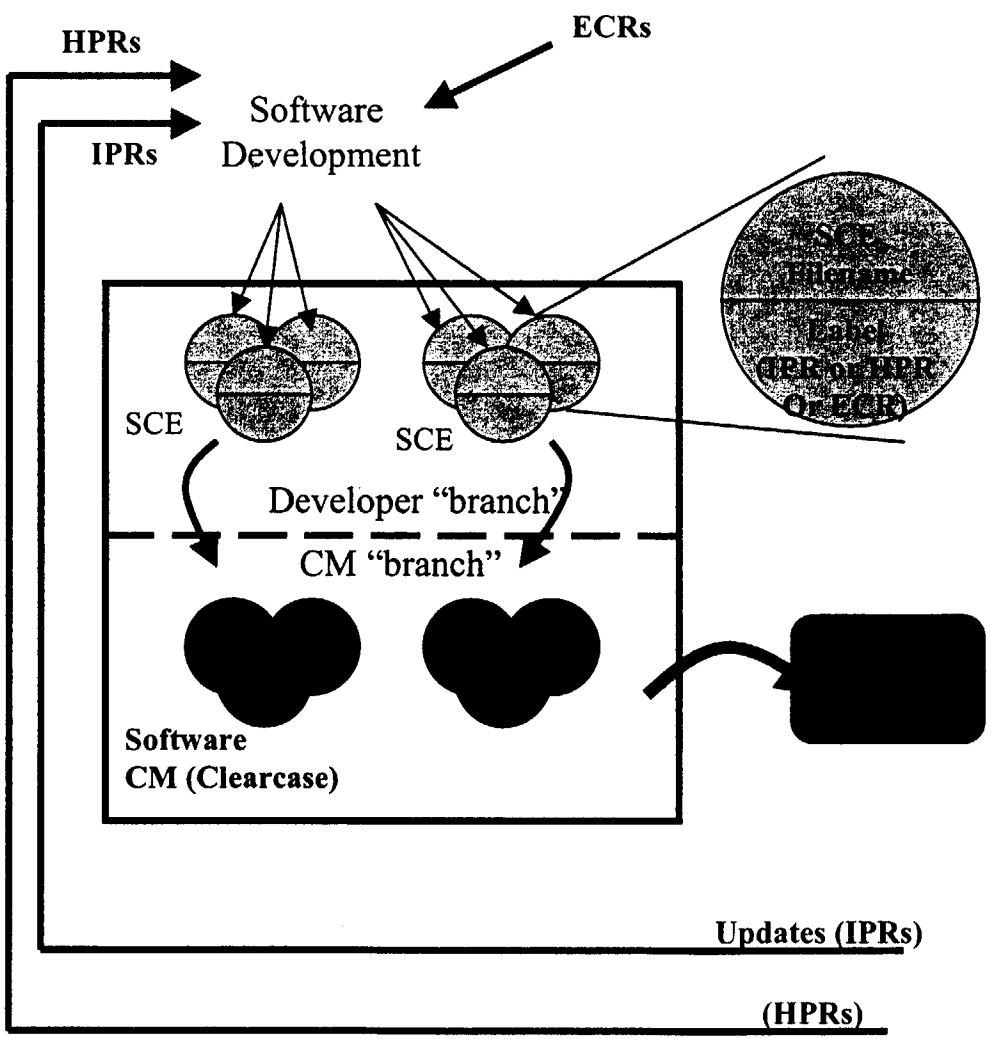
Software
Change Control Process



Software CM and Build

- Software CM is the key to making accurate and timely updates to the system baselines
- HPRs are written by just about anybody, and they are approved by the HPR Board. The HPR Board consists of representatives from all Stake Holders
- IPRs are written by DIT Testers
- ECRs are written by Systems Engineering and approved by the HMCG, which is also made up of Stake Holders.
- The CM Organization creates Code Baseline Checklists that use to generate "merge" requests
  - These requests are submitted when source code is ready for the CM build
- Software CM then "makes" the EHS build
- Filemaker Pro is and important COTs product in this process

Software Change Control (Labels)



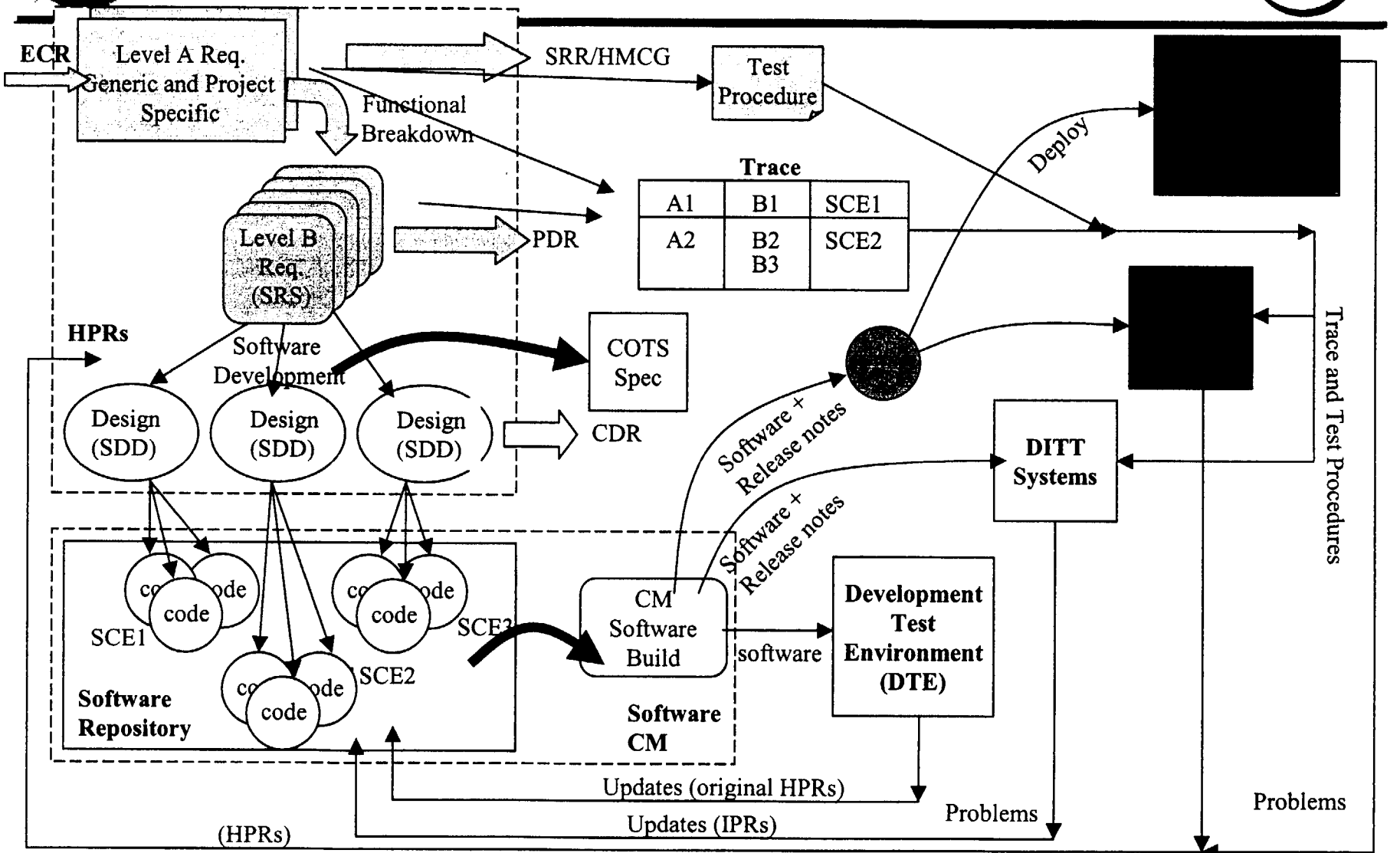
Software CM and Build

- The source files that make up an SCE are "tagged" by the developer with a label consisting of the ECR, IPR, or HPR number
- Tags are available only for approved changes..
- Code merge requests are also only take for approved changes
- CM merges (creates a new version) the files to the CM branch and then makes the binary images

Spiral Development in EHS

- Spiral Development Model
  - "Risk driven model used to guide multi-stake-holder, concurrent engineering of software intensive systems
    - POIC stake-holders: CADRE, FD, Testing, Development, System Engineering
  - Main features of Spiral Development
    - Cyclic approach for incrementally growing a system's degree of definition and implementation
    - A defined set of "Anchor Point" milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions
  - Advantages of Spiral Development Model
    - Avoids premature commitment to requirements, design, COTS, cost/schedule
    - Avoids "analysis paralysis"

Spiral Development Cycle (yellow)



Spiral Development in EHS

- For EHS the end-of-life of our hardware platforms has driven us to make some major changes
- On the Software Development and Test drawing we chose specific Level A's we wanted to implement on new platforms, and we began prototypes in those areas
- As those areas succeeded, we chose more requirements and thus gradually grew, or spiraled the migrated system requirements.
- So we updated the trace and added the items shown in red "e1.0" etc…

Spiral Development Updated Trace

027v1 Par	SRS 027 V1 Requirement	B Bld	Lev A Doc	Lev A Par	Level A Requirement
3.2.2.1.a	The Interface Display Operation User process shall adhere to MSFC-STD-1956.	2 | 3 | 4 | 4.1 **e1.0**	MSFC-RQMT-1440	14.2.1.A	The Display Operation UI shall adhere to MSFC-STD-1956.
3.2.2.1.aa	The Interface Display Operation User process shall provide the capability to enable limit/expected state sensing for all objects on a display.	2 | 3 | 4 | 4.1 | **e2.0**	MSFC-RQMT-1440	14.2.1.L	The Display Operation UI shall provide the capability to toggle limit sensing on and off.
3.2.2.1.ab	The Interface Display Operation User process shall provide the capability to zoom or un-zoom a time or XY plot on a display.	2 | 3 | 4 | 4.1 **e2.0**	MSFC-RQMT-1440	14.2.1.C	The Display Operation UI shall provide the capability to view data as updated.

Lessons Learned

- Avoid over-analyzing requirements
- Do a more iterative development approach with user involvement
- Don't waste man-power on document details
- Make sure your test environment looks like your operational environment as much as possible
- Remain open to continuous process improvements
  - We automated the software change control process
  - We streamlined the software Design documentation process
  - We allowed for the "Spiral" approach in our PIMS and PC Migration

Conclusion

- Software development and test is a job of managing change
- Too much "process" and you never deliver
- Too little "process" and you deliver low quality products and that takes time to fix, and you never deliver….
- We have evolved EHS process to inlcude some old, and some new ideas
  - Traditional handling of requirements
  - Newer concept of iterative development with user involvement and proof of concept
  - Automated keeping track of defects and correlating defects to fixes via the software cm processes
  - Feedback test metrics into the process for continuous improvement
- We have been able to "changed" the way we manage change

Software CM Terminology

- **make**: the process of compiling and linking software source files
- **build:** the process of running the make and creating binary images in the form of executables and libraries that are combined into a release package that is loadable.
- **load:** the process of taking a successful Build and installing it to run on a computer
- **merge:** the process of moving a source file from one are of control called a "branch" to another area or branch. When a source file is merged, a new version of the file is created in the CM tool on the branch where the file is merged to.
- **branch:** and area of control. Developers have branches, groups of developers have branches, and the Software CM group has branches.
- **version:** an instantiation of a source file