SLAC CONTROLS GROUP

SOFTWARE REQUIREMENTS AND ANALYSIS METHODOLOGY

RECOMMENDATIONS FOR NLC CONTROLS

Greg White 28th June, 2000, SLAC
Rev 1.32

RECOMMENDATIONS FOR NLC SOFTWARE REQUIREMNTS AND ANALYSIS METHODOLOGY

Rev 1.1: Changed Db requirements in appendix 1 “RequistePro Requirement Types”

Rev 1.2 Changed introduction to remove estimated total cost.

Rev 1.3. Added Executive Summary

1 EXECUTIVE SUMMARY OF THE METHODOLOGY

The NLC controls project will take many years, involving a collaboration of many engineers, but the staffing level will not be constant. Our short term goal is to accurately estimate costs, decompose the problem, and to clearly demonstrate where we are. In the long term our goal is to move smoothly from functional requirements to specific sub-system or component requirements for individual, geographically distributed programmers. This methodology covers both ends of that lifecycle. It covers both analytical and documentary tools for software engineers, and project management.

A clear statement of the deliverables will clarify all stakeholders expectations, so tables of the deliverables are presented explicitly here (see Table 1, Table 2, Table 4 and Table 6). If this recommendation document is approved these will be expanded so that its clear what each individual job function is responsible for; and workflows will be drawn up for each work function.

This methodology distinguishes between very general requirements such as the "goal" and overall functional "needs", and those which are more specific, possibly testable items, which we call "features" and "system requirements". As such, requirements can be viewed hierarchically, from the goal to the system requirements (see Figure:1) The methodology includes a framework for analyzing all levels requirements.

Requirements are seen as having attached "attributes" such as benefit, effort, cost, time, priority etc. The methodology concretizes that notion, and presents a way to document all of those attributes for each individual requirement (see 5.3). It's shown how those attributes are used for project planning.

In a large project there will be large number of alternative requirements and solutions proposed, each impacting the requirements of associated systems and changing their cost and time estimates. The methodology manages "alternative requirement sets" (see 5.3.3 and 5.3.4) and can manage the treaty points between systems to minimize friction and scope creep (see 5.4).

A software development tool, Rational RequisitePro is presented, which can help visualize the requirement dependencies, the alternative requirements, and their attributes, to help both project management and developers (see 3.1.3).

"Use cases" are used to delineate requirements at the levels of Features and Requirements (see 4.2.5). The development cycle proposed is iterative and "feature driven", which attempts to formalize the rapid prototyping approach so that it's a procedure which can be effectively managed across a number of development centers. Use cases, and iterative feature driven development, will help collaborative development.

Its recognized that in a functionally decomposed control system, the software infrastructure, and the network architecture, will also give rise to important requirements and constraints, though they will not be functional requirements as such. Those requirements will also need to be analyzed and managed within the methodology (see 4.3). Utilities and general purpose software components will be built to satisfy those requirements, which will expedite the development of software which satisfies the functional requirements. The methodology does not enforce Object Oriented design but it does support it and encourages a component based software framework.

Recommendations of so called "systemic systems analysis" are also promoted in the methodology. These address issues of the system as a whole, and "human factors" within it.

2 INTRODUCTION

In this document we detail recommendations for a methodology for the Requirements and Analysis tasks involved in the development of controls software for the Next Linear Collider (NLC). The NLC will be a large electron-positron particle accelerator which a collaboration led by Stanford Linear Accelerator Center, together with Lawrence Livermore National Laboratory (LLNL), Lawrence Berkeley National Laboratory (LBNL), and Fermi National Laboratory (FNAL). The collaboration may also include other laboratories in the US or abroad.

2.1 OUR SITUATION

The estimated cost of the NLC is A LOT OF MONEY, of which about $0.5bn is for controls. The software development effort and support infrastructure is about $75m, and represents ~475 person years of effort. However the funding calendar requires that our staffing is not constant over the development period. Additionally, the software controls effort is likely to be collaborative. In particular we hope to use developers of the EPICS control system framework, although the extent to which we will use EPICS, and the basis of our collaboration in the EPICS community, is not yet clear. Therefore this methodology addresses:

- Communicating future intent and alternatives
- Defining the “treaty points” between sub-systems, and the interfaces between sub-systems
- How sub-systems relate to the network architecture and data infrastructure (such as data acquisition, control, database and archiving).

2.2 OBJECTIVE

The objectives of our requirements definition efforts, and therefore the methodology we employ are significantly different over time:

- In the short term the objective of our methodology is to help us effectively estimate the software and infrastructure requirements for the purposes of cost estimation and project scheduling.
- In the long term our goal shifts to producing requirements suitable for producing accurate and clear designs.

This methodology is designed to help on both fronts.

Additionally, the methodology separately outlines development and management methods.

Clearly writing documentation is not the same as analyzing requirements, but in describing the deliverables, how they relate to each other, and how they’ll be managed, we hope to describe a process which can be easily followed to get results. The intention is to outline techniques rather than to describe them, and to give references to full explanations in existing literature.

2.3 SCOPE

This methodology views the process of requirements analysis as being ongoing, through development, testing and maintenance. That is, it's a largely iterative scheme, based on the Rational Unified Process. The need to fit our system into a larger architecture, test software, maintain it efficiently, and to build in cost minimization via change control and risk analysis, means that we add to the nominal functional "requirements" further specifications for:

- **Test and Quality Assurance Requirements**. Passing all the test cases will itself be a requirement of the system. Each test case will be made up of specific tests, which will be documented as part of the requirements.
- **Change Control Requirements**. The impact of change requests made during development must be made explicit. This is achieved by presenting system functions through Use Cases and through "traceability" (see Traceability3.1.2).
- **Risk Analysis Requirements**. With such a long development time-span, the number of different systems that must be developed, and their technical sophistication, there is a considerable risk of over-run. This is addressed by using risk "attributes" attached to requirements, and the development of alternative sets of requirements and designs.
- **Collaboration Support and Communication Requirements**. How are intentions, technology, and changes to designs and software to be shared among us? This will be achieved through Use Case modelling, maintaining a comprehensive Glossary and References, interaction and collaboration diagrams, and maintaining an automated requirements management system based on Rational RequisitePro.

The methodology is intended to support a multi-level, component based object oriented design approach.

This document describes recommendations for the methodology. When we've decided on the details we should convert this document to guidelines for that methodology.

2.4 "REQUIREMENTS" AND DESIGN

In common usage sometimes the word "requirement" is used to mean a general item of description about a system, such as that it "must help you look at BPM data", and sometimes a very specific testable item, such as that "on z plots the unit number of e- only BPMs must be displayed in gray on a e+ only BPM plot". This illustrates a "life-cycle" since the former kind of requirement is specified typically first, and the second much later. This methodology covers both ends of that requirement life-cycle. It proposes a hierarchy which distinguishes high level from low-level requirements, but emphasizes that there is a "traceability" between them – high level requirements own low level ones, and ultimately high level requirements are only satisfied when all of their low level requirements are tested and working. We will use the Rational Unified Process hierarchy, which divides "requirements" into three levels; Needs, Features and Requirements (see Figure:1Requirements Hierarchy below). In this document, the word "requirement" is used sometimes in its broad sense, and sometimes in the specific sense of an atomic, testable item.

2.4.1 SCOPE OF REQUIREMENTS AND DESIGN

This methodology assumes that a major design principle will be to use component or framework based OO programming to maximize code reuse and simplicity of design. Therefore the components, middleware, frameworks and libraries we produce will themselves be specified by requirements, independently of the eventual overall system design. Management of those requirements, which are motivated by the design process, is included in this methodology.

Also, to support distributed collaborative programming, we need to be able to closely specify the program details, such as the color of the e+ BPM unit numbers, so we include that kind of very specific requirement. .

2.4.2 INTERFACE OF REQUIREMENTS TO THE DESIGN PROCESS

The methodology includes directions and recommendations up to and including the specification of a Domain Model. This includes the specification of Use Cases, Supplementary Specifications, and if relevant Activity Diagrams and Interaction Diagrams. Those are the stages immediately before Class Collaboration Diagrams are produced in the Object Oriented Analysis and Design methodology. The Domain Model should feed directly into the Class Collaboration scheme, and result in direct creation of code stubs. The association between this methodology and the design process would be explained further in the Design Methodology, if we ever write that.

2.4.3 LEVEL OF BENEFIT– MIGHT, COULD, SHOULD, WILL, MUST

Given the scale, longevity, and the number of inter-related systems, of the NLC control system development, it seems sensible to record all our ideas and possible solutions rather than only those which have been absolutely certified as “must haves”. This is because we have to be able to explore alternatives early on in development. Therefore, we will include a formalism which allows us to put together alternative combinations of subsystem requirements, which can take account of items which might, could, or should be included, but help us estimate the impact on scope and cost creep (see 5.1 below). This scalar “benefit” attribute of a requirement is one of a number, including priority, status, and stability, which will help us schedule work.

2.5 “METHODOLOGY”

Ok, “methodology” strictly speaking means the “study of method”, but its meant here in its now common usage as a description of a method.

This methodology is largely a combination of the following:

- The Rational Unified Process (R.U.P), particularly the Requirements and Analysis using Use Cases (R.A.U.C), Rational 1998
- Object Oriented Analysis and Design (OOAD), various sources, Liberty 1998 in particular.
- General Systems Theory (GST), Bertalanfy 1968.
- Various Systematic approaches, in particular Systems Engineering and the “Traditional Approach” (sometimes called “Waterfall”).
- Various systemic systems analysis methods (those which emphasis the system as a whole in its environment), in particular Human Activity Systems (Mumford et al) and the Participative Approach.

2.6 DELIVERABLES

The deliverables of this methodology are fully described in the relevant sections of this document (see sections 4, Requirements and Analysis for Development and 5, Requirements and Analysis Management and Architecture). However, we summarize them below (see Table 1). Additionally a suggested set of RequisitePro "tags" to support this method, and template documents, are given in the Appendices.

Items in gray are not discussed further in this document. They will form the basis for the Design Methodology if we create one. To learn more about Activity Diagrams see Liberty 1998, for Interaction Diagrams see Liberty 1998, Rational 1998, any UML book.

Table 1: Summary of Deliverables for each application or component library. See Table 4 for details.

Artifact Group	**Artifact**	**Description**	**Authors**
Vision	Vision Document	Overall summary of purpose and desired features	Champion, Engineer and Physicist users, Analyst
Functional Requirements	Domain Model	Use Case Model: Diagrammatical and **Textual** description of Use Cases (that is, interfaces). Entity-Relationship diagrams. Algorithms	Analyst, Use case Specifier
	User Interface Model	User interface mock-up	Champion, Developer
	Use Case Package Report:	Diagrams of relationships of packages and subsystems	Inception and Elaboration Phases
Supplementary Specifications	Non-functional requirements Document	Textual description of system constraints and references	Analyst, Champion
	Conflict Matrix	How conflicting data acq or control requirement will be handled.	Users, Champions, Architect, Analysts.
	Taxonomy	Summary of key functional attributes	Architects, Designers
	Requirements Matrices	RequisitePro db of requirements	Designers, Developers
	Cost Benefit Analysis	Description of Cost, Effort and functionality alternatives and estimates	Managers, Architect, Designers, Developers
Requirements and Design	Activity Diagrams	Defines synchronization of process activities	Architect, Analyst and Designer
	Interaction Diagrams	Shows how objects interact with each other and external objects of other systems	Architect, Analyst and Designer
	Class Collaboration Diagrams	Shows Class Schema, of this subsystem and external objects of other systems	Analyst and Designers
Test	Test Specifications	Details Test Cases	Champions, Designers, Developers
Commissioning	User Document Specifications	User Guide	Champions, Developers

Table 2: Summary of Requirements Deliverables for Management and Architecture. See Table 6 for details)

Artifact	**Description**	**Authors**
CDR summary	Summary of requirements work breakdown	Management, Architect
WBS	Work Breakdown Structure	Management
Developers Guide	A combination of the SLC style BUG and POOP. Description and APIs of core systems	Architect, Designers, Developers.
Basic Design Principles	Statement of intent about the core architecture, infrastructure, and design of systems.	Management, Architect, Designers, Champions
Sizing and Performance Specification	Diagrams and text showing nodes, data flow, and timing requirement	Champion, Network Architect, Analysts
System Boundary Analysis	Summary of System Boundary and Treaty Points	Managers, Architects, Designers
Domain Analysis	Document and Diagrams detailing domain objects and their relationship	Architects, Analysts, Champions, Designers
Enterprise db Schema	Description of control system entities and their relations	DBMA, Architects, Analysts, Designers
Glossary	A dictionary of terms.	Everyone
References	Pointers to literature, both accl. Physics and Engineering and programming	Everyone

2.7 WHO'S WHO

The activity roles used in the deliverable table map roughly to the following job descriptions at SLAC.

Table 3: Job Descriptions

Functional name in this Methodology	**Who at SLAC**
Users	Physicists hoping to use the control system
Management	Those responsible to NLC executive management
Champion	Physicist or Engineer volunteered to champion a subsystem's development. Virtual Tom
Analyst	Developer
DBMA	Developer with special interests in DB design
Use Case Modeler	Developer and Champion

Designer	Developer
Database Designer	Developer, DB Administrator
Documentation Writer	Developer
Tester	Developer, Champion, Operator
Architect	Developer and Architecture group

“Developer” is user here to mean the primary software writer, probably and Application Developer, Specialist or Supervisor in the Controls Software group.

3 METHODOLOGY IN OUTLINE

The key features of the methodology are described in this section. In particular the hierarchical nature of requirements when viewed over the lifecycle of a project, that such requirements are multiply connected and how we can manage those relationships, and a proposal for a formalized iterative development approach.

3.1 REQUIREMENTS HIERARCHY

Overall our goal is to deliver good software on time and in budget, which solves the root problem of the users, by satisfying their real needs. These "needs" should therefore be acquired *explicitly*, so as to minimize scope creep, and therefore cost creep, rather than being merely implied by the itemized requirements. The hierarchy of requirements therefore reads:

- **Purpose**. The problem statement.

- **Needs**. These are the overall descriptions of a software system's objectives. For example the BPM display software must help a user "browse real-time BPM data". From there the requirements process shall also include translation of those needs into some set of

- **Features** satisfying the needs, similar to the sort of features that might be advertised for commercial software.

- **Requirements**. Lastly the individual software functions which satisfy those needs and features are itemized. These are atomic, and testable. The requirements process is integrated with the design process to some extent, and there is an iterative refinement between them

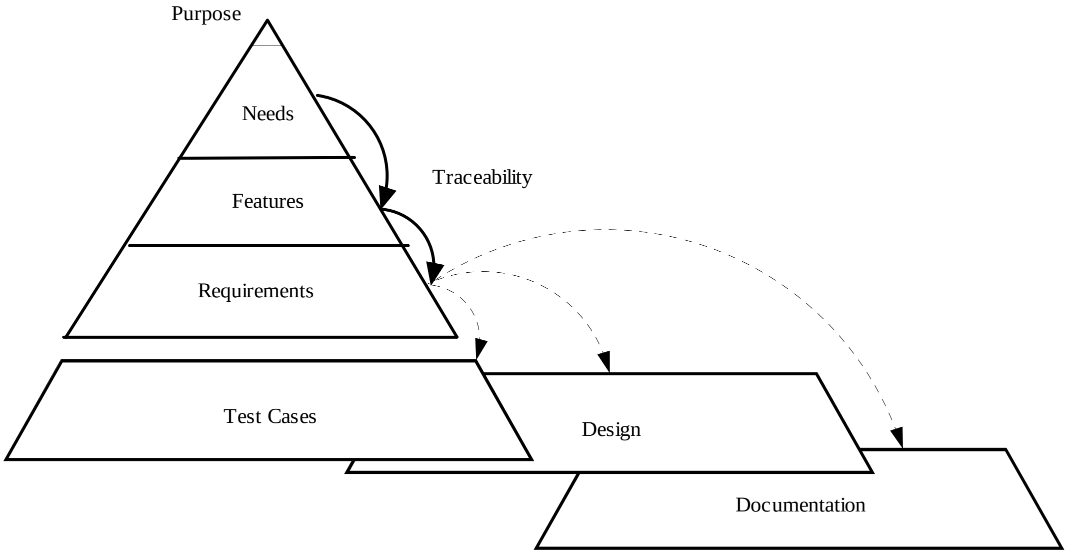
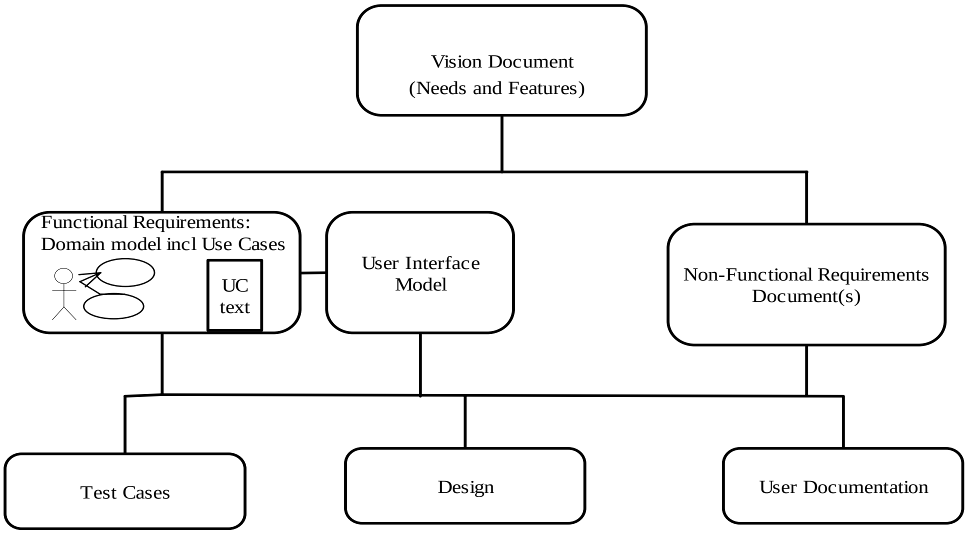


Figure:1Requirements Hierarchy

The real needs of our users may be hidden by our desire to use previous experience and familiar technology. For instance, we don't have a requirement to produce a BPM Sampler, we instead have a need to make available per-pulse data from BPMs.

3.1.1 WHERE NEEDS, FEATURES AND REQUIREMENTS ARE DEFINED

The Needs, Features and consequent Requirements of each software subsystem are detailed in the core documents of the requirements process for development (see below). Those core documents are:

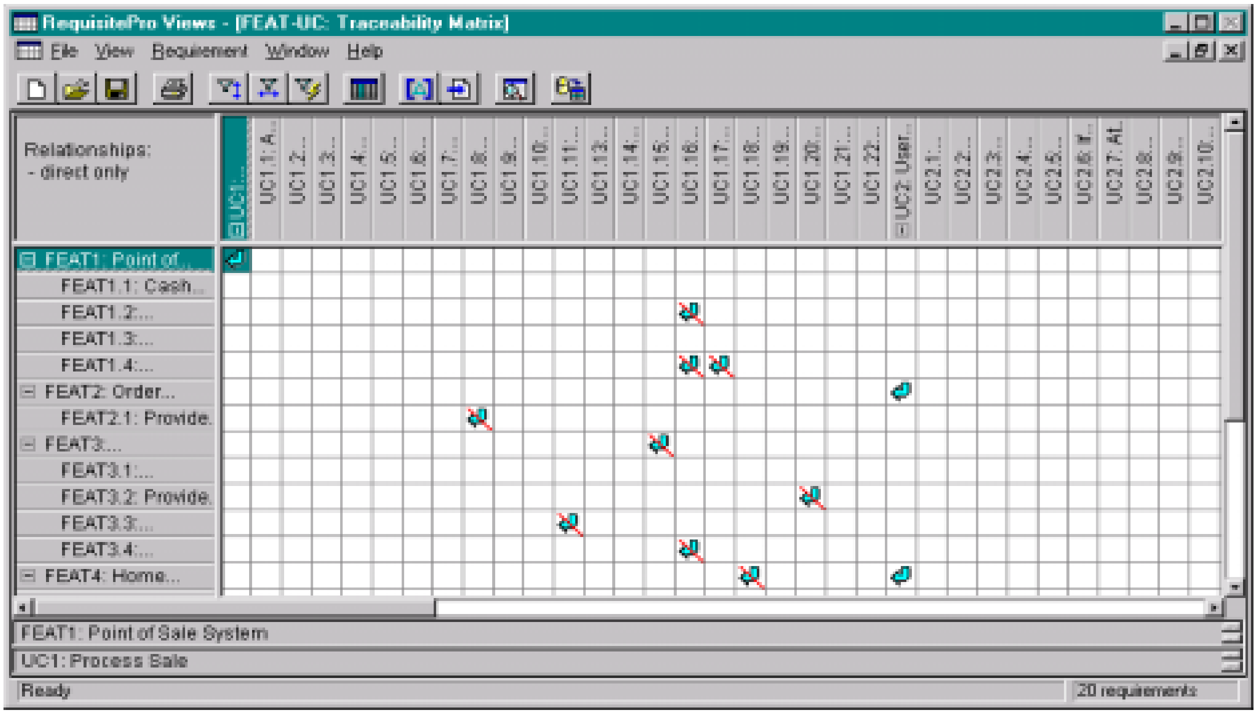


- Vision Document: This will include the **Needs** and **Features**.
- Requirements Specifications
    - Domain Model: this includes the Use Case models, which contains **Functional Requirements** in the textual descriptions of the Use Cases.
    - User interface model, and possibly prototype.
- Supplementary specifications: This includes **Non-functional Requirements**, such as system constraints, performance constraints, sub-system collaboration constraints etc.

3.1.2 TRACEABILITY

The hierarchical relationship of *needs* to *features* to *requirements*, and their mutual implication, is called “Traceability” in the Rational Unified Process. Rational’s RequisitePro product can be used to manage and report the traceability relations. (see Figure 2). In this way each requirement is traceable to a need which motivated it, and downwardly traceable to a test case which can verify that it has been meet.

Traceability is also a foundation for Object Oriented development. Object technology relies on linking discrete object models together. Creating clear “small” objects for each small task has been found to be a prerequisite for realizing the promise of reusable, rapidly developed code. Each small object should trace back to one or more small clear requirements, and the requirements to larger features.



RequisitePro Views - [FEAT-UC: Traceability Matrix]

Relationships: - direct only	UC1...	UC1.1: A...	UC1.2:...	UC1.3:...	UC1.4:...	UC1.5:...	UC1.6:...	UC1.7:...	UC1.8:...	UC1.9:...	UC1.10:...	UC1.11:...	UC1.13:...	UC1.14:...	UC1.15:...	UC1.16:...	UC1.17:...	UC1.18:...	UC1.19:...	UC1.20:...	UC1.21:...	UC1.22:...	UC2: User...	UC2.1:...	UC2.2:...	UC2.3:...	UC2.4:...	UC2.5:...	UC2.6: T...	UC2.7: Al...	UC2.8:...	UC2.9:...	UC2.10:...
⊟ FEAT1: Point of...	↵																																
FEAT1.1: Cash...																																	
FEAT1.2:...																↵⧸																	
FEAT1.3:...																																	
FEAT1.4:...																↵⧸	↵⧸																
⊟ FEAT2: Order...																							↵										
FEAT2.1: Provide...									↵⧸																								
⊟ FEAT3:...															↵⧸																		
FEAT3.1:...																																	
FEAT3.2: Provide...																				↵⧸													
FEAT3.3:...												↵⧸																					
FEAT3.4:...																↵⧸																	
⊟ FEAT4: Home...																		↵⧸					↵										

FEAT1: Point of Sale System

UC1: Process Sale

Ready

20 requirements

Figure 2: RequistePro Traceability matrix showing relationship, in this case, between Features and Use Cases. Those marked with a diagonal bar show requirements which have changed, and therefore whose association should be rechecked by the author.

3.1.3 REQUIREMENT TYPES AND ATTRIBUTES

Rational Software’s RequisitePro is a tool for managing requirements. It defines a “requirement” in its database in a very general way - as a *name/value* pair. For instance two types of requirement that RequisitePro comes with defined are named “FEAT” for Feature, and UC for Use Case. Figure 2 shows an example of the traceability between these two attributes, “FEAT” and “UC”, for a hypothetical project. The values of these requirements are the text in the documents that defined them.

Other feature types can be invented by the analyst. For instance one could add a type named DBENTITY for a requirement for a database entity. Some more concrete example uses of this ability to track and cross reference objects to requirements are given below. Each of these will be implemented as a RequisitePro requirement type. A proposed list of these is given in Appendix 6.1.

- Data Base Entity and Control devices. We can track which FEATures require which db entities.
- Networks (which network carries the needs of which project for instance)
- Panel, or GUI object (which panel implements a particular requirement)

- Glossary. RequisitePro comes with a GLOSS requirement type for associating glossary definitions with other requirements.

Each type describes requirements by their attributes, such as Cost, Priority, Status, AssignedTo etc. RequisitePro also allows us to easily add attributes to the list it can manage. So RequistePro can be used to track a very broad range of information relevant to a project.

Additionally, support for design, user documentation, and collaborative communication are supported by the traceability and attribute functions (see section 5).

Obviously not all requirements “trace” to a need. For instance, the treaty points and system boundary are absolute requirements. Those can be managed by RequisitePro too.

3.2 ITERATIVE APPROACH

The methodology we propose is iterative and feature driven. In contrast to the Traditional, or “Waterfall” approach where requirements are defined carefully before moving on to design, and design is completed before coding, implementation and testing, our approach will allow software to be grown incrementally. However, in contrast to the similar Rapid Prototyping approach, where features are supposed to be discovered and coded in a sort of creative frenzy, this one uses quite strict guidelines for the deliverables, workflow, and oversight.

For our purposes this structured iterative approach is intended to:

1. Resolve hard problems and critical issues of system collaboration early
2. Ameliorate the risk of project overrun inherent in very long lead times, by setting short term goals
3. Help with system integration, since the point at which one subsystem is ready to service another can be brought forward, rather than waiting until both are complete
4. Allow and capitalize on user feedback early
5. Improve the measurement of progress
6. Allow partial system deployment.

This iterative model is based on the Rational Unified Process:

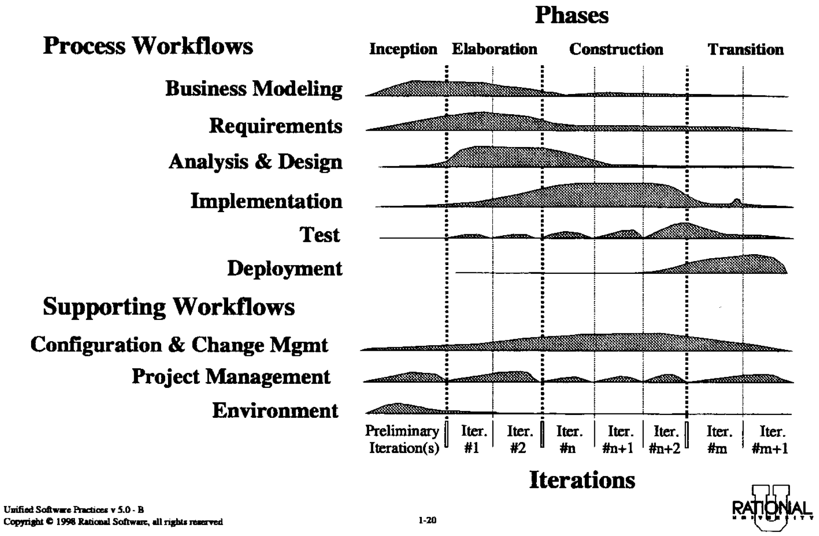


Figure 3: Iterative Life Cycle. Some fraction of all the usual stages of system development are performed in every phase.

3.2.1 FEATURE DRIVEN

As a modification to this approach, our approach will make the iterations within each phase, of fixed duration. Additionally, the intended features to be delivered at the end of each iteration will be fixed. So, rather than making the number of features, and the overall deadline fixed, as in the traditional system development approach, in this one the number of features to be undertaken in each iteration is variable, but the time to each milestone is fixed. This is to help the management of a distributed collaboration (so everyone knows the next waypoint, which may include a system integration) and to get a sense of purpose and achievement out of each feature set as its delivered.

Sets of features can be grouped into desired “feature sets” for each release cycle. The progress of each feature set can be tracked, rather than each individual requirement, while at the same time we are able to sit back and re-evaluate our approach without waiting until the whole system is finished before we decide to dump it and go to the pub.

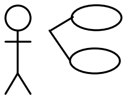
The “traceability” tool discussed above helps us to chose consistent feature sets to implement at each iteration because their interdependence is documented.

Cost-estimation also benefits from managing each development cycle as a set of features, see 5.3 Cost Benefit Estimation.

4 REQUIREMENTS AND ANALYSIS FOR DEVELOPMENT

This section describes the principal deliverables of the requirements analysis which are the responsibility of the development team. Table 4 lists those deliverables. Here we describe the Vision Document, the Functional Requirements (including the Use Case modelling and User Interface modeling), and the Supplementary specifications. Also some specific tools and techniques for doing the analysis are given. The items in gray are not discussed further. They would be elaborated in the design guidelines.

Table 4: Deliverables for Each Subsystem

Artifact Group	**Artifact**	**Description**	**Authors**	**Mainly When**	**Contents**	**Audience**
Vision	Vision Document	Overall summary of desired features	Champion, Engineer and Physicist users, Analyst	Inception	Needs and desired Features. Alternative system outlines. Noted technical difficulties	Uses, Management, Architects, Analysts, UC specifiers, Designers, Testers
Functional Requirements	Domain Model	Use Case Model: Diagrammatical And **Textual** description of Use Cases (that is, interfaces). Entity-relationship diagrams Algorithms	Analyst, Use case Specifier	Inception and Elaboration Phases	 UC Names, package, Scenarios, Preconditions, Postconditions, Constraints, Related vision, and non-functional Spec.	Champion, Design review, to **validate understanding**. Documentation Writers. Testers
	User Interface Model	User interface mock-up	Champion, Developer	Inception and Elaboration	Prototype GUI	Champion, Users, Designers
	Use Case Package Report: Diagram of relationship of packages and subsystems	Analyst, Use Case Specifier, Architect	Inception and Elaboration Phases	Package name, Set of UCs, including Actors, objects, relationships, e.g. <<uses>>, <<extends>>	Architect, to validate how sub-systems fit together.	Designers, Developers, Management

Supplementary Specifications	Non-functional requirements Document	Textual description of system constraints and references	Analyst, Champion	Inception and Elaboration	**Performance** requ., System Requirements., Architectural Spec., Design Recommendations, Forward References	Architects, Analysts, Designers,
	Conflict Matrix	How conflicting data acq or control requirement will be handled.	Users, Champions, Architect, Analysts.	Inception and Elaboration	Text and definition of which system takes priority in a conflict	Users, Champions, Designers
	Taxonomy	Summary of Key attributes	Architects, Designers	Inception and Elaboration	Taxonomy table: When Initiated, Frequency, latency, synch time etc.	Everyone.
	Requirements and Traceability Matrices	RequisitePro db of requirements	Designers, Developers	Inception and Elaboration	Requirements hierarchy and Traceability matrices	Designers, Developers, Managers
	Cost Benefit Analysis	Description of Cost, Effort and functionality alternatives and estimates	Managers, Architect, Designers, Developers	Inception and Elaboration	Quantitative cost/effort analysis matrix.	Managers, Champions
Requirements and Design	Activity Diagrams	Defines synchronization of activities	Architect, Analyst and Designer	Elaboration	UML Activity Diagram, good for real-time requ. Spec.	Architect, Designers
	Interaction Diagrams	Shows how objects interact with each other and external objects of other systems	Architect, Analyst and Designer	Elaboration	Either Sequence diagram or collaboration diagram, feeds into Class collaboration	Designers, Developers
	Class Collaboration Diagrams	Shows Class Schema, of this subsystem and external objects of other systems	Analyst and Designers	Elaboration	UML Class collaboration diagram	Designers, Developers
Test	Test Specifications	Details Test Cases	Champions, Designers, Developers	Elaboration and Construction	Test Cases and Specific tests which will validate a sub-system	Testers
Commissioning	User Document Specifications	User Guide	Champions, Developers	Construction	How to use the system	Users, Testers

4.1 THE VISION DOCUMENT

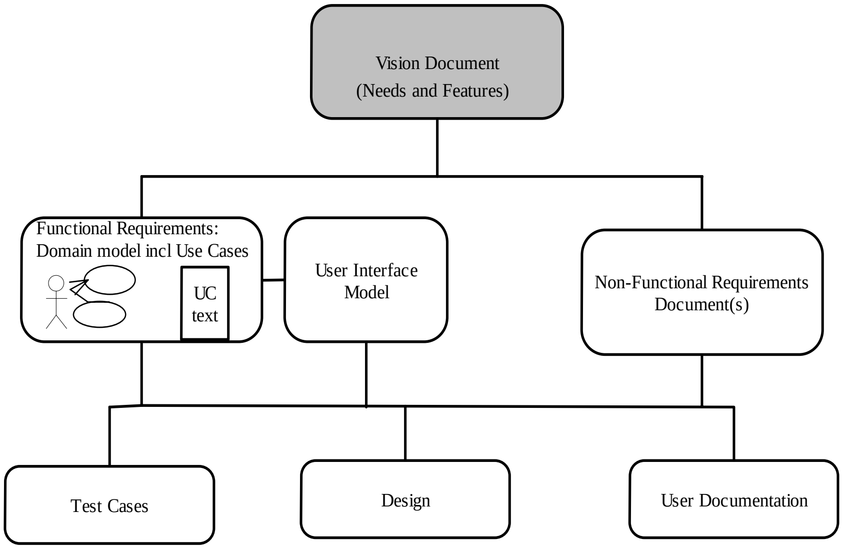


Figure 4: Where the Vision Document fit in to the core deliverable hierarchy

As the name suggests the vision document will focus on the intention of the system. It must also include a description of *why* the system is needed, so that stakeholders can adequately review it. Its role is to communicate desires between management, our users, and developers both at SLAC and in the software collaboration. A template for the Vision Document will be decided on.

In outline it shall contain:

- Purpose - the basic statement of the intention of the system. Whenever possible this should not be phrased in terms of the requirements of other systems, but rather in terms of users' real needs. General Systems Theory calls this the "Root Definition", and stresses that it should be phrased independently of tradition or existing systems, historical bias, or artificial pressures from the requirements of other systems [Bertalanaffy, 1968]. For instance, consideration should be given to whether a system is being requested only because SLC or PEPII had this system as a component in their control system, rather than the core requirements of NLC's control. The RUP includes a template phrase for this as a "position statement".

- Goals and objectives
- References to the Functional and Non-functional requirements documents, to the vision statement of related subsystems, to the RequisitePro db, to the Sizing and Performance document, to the System Boundary Analysis, and to the Domain Analysis.
- Users **Needs**. These should be itemized, with supporting use cases.
- Alternative System outlines (see 5.3.3)
  - o **Features**, ordered by alternative outline

- Target audience of the system. A description of the users or list of client and server subsystems which will interact with this subsystem. This will form the basis for the **actors** in the Use Case Model.
- Environment, platforms and other constraints, such as how this system fits in with other systems, the data infrastructure, and architecture.
- Noted technical Difficulties
- Record of future features and ideas
- Other requirements, such as applicable standards, portability and extensibility requirements etc. Performance constraints are listed separately in the Supplementary Specifications and the Sizing and Performance Specification (see and Table 1).

4.2 FUNCTIONAL REQUIREMENTS

The functional requirements are composed of the Domain Model (the Use Cases and the description of the problem domain) and the User Interface Model.

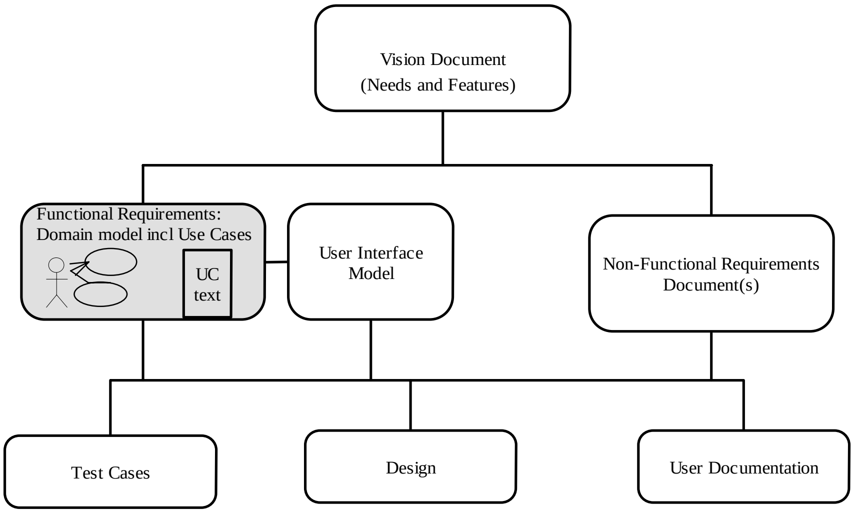


Figure 5: Where Functional Requirements fit in to the core deliverable hierarchy

4.2.1 DOMAIN MODELING WITH USE CASES

A “Use Case” literally describes one case of the use of the system. That is, it makes explicit one instance of an interaction between the system and the user (or the interaction may involve more than one user). A use case is a flow of events between the system and a user. Rather pompously, the literature says that normally a Use Case is taken to mean “an interaction which leads to an observable result of value to the user”. The Unified Modeling Language (UML), from which Use Cases are taken, calls a user an **actor**.

The actor is not necessarily an individual – it may be another sub-system. Additionally, an actor is distinguished from simply a user or sub-system in that an actor specifies one role. Each process an actor must go through to fulfill one role defines one Use Case of the system.

So for instance a Use Case is not an instance of one button push, rather one Use Case might describe setting up and viewing a BPM orbit plot. Each use case is a set of actions and the actions should be atomic (either performed in their entirety or not performed).

Use Cases might be used to specify requirements and processes at a very high level, such as in Business Modeling (which isn't discussed here), at the stage of describing Needs and Features, at the Requirements specification stage, and at the Design stage. At each level, a different level of detail is required, and more actors will come into play. In the BPM orbit plot example, if this were a Use Case at the Requirements elicitation stage, there would only be one actor - a "Control System User". At the Design stage this Use Case will involve one or more actors for the data acquisition as well. Here we concentrate on Use Cases for Requirements.

Requirement Type	**Use Case Detail**
Purpose	Use cases are not used at this stage
Needs	High level, "business model" use cases. This application of use cases is optional
Features	Use cases named and **elaborated** (a short paragraph description); **scenario** written (see Figure 7); main **actors** identified.
Requirements	The **use case description** is itemized (see Figure 8). Each use cases is given in context to others, with <<uses>> and <<extends>> where appropriate.

Table 5: Where use cases are used, and at what level of detail

Use Cases should not typically be arranged hierarchically! This is especially true of Use Cases used to specify requirements, rather than design. A Use Case may be quite a lengthy description of an interaction, and involve back-and-forth dialog with the actor. It tells you how the actor interacts with the system and includes how the system responds.

The Use Case Model of a system is all of the actors and all of the various use cases. In principle the complete set of use cases for a system forms a complete description of the functional requirements of the system. However, that may turn out to be too idealized for us.

Use cases are very thoroughly described in the literature [Rational 1998, Liberty 1998, Weigers 1999]. We describe here some common recommendations.

4.2.2 IDENTIFYING THE USE CASES

In general developing a use case isn't hard, but making sure you have all the use cases is hard. The literature warns that use cases are sometimes so obvious that they're overlooked. Sources for developing use cases are:

- Domain experts.
- Existing PEPII and SLC software (obviously being careful not to reproduce behaviour which is not optimal for he NLC).

- Users, operators.
- Participative analysis. For user level software its particularly useful to sit in the control room to see first hand what's done, what takes time, what could be automated etc [The Participative Approach, Mumford et al]

4.2.3 IDENTIFYING THE ACTORS

Clearly the basic step to finding the actors is to consider how the system will be used. Other tips from the literature are:

- Ask the domain experts to define who performs the main tasks with the system, and what tasks do they perform.
- Who will perform the secondary tasks, and maintenance tasks?
- Which other systems need information from this system. Which other systems does this one send information to?
- How is the system initialized. How is it terminated?
- Is there a log on, or log off sequence?

Document the names for the actors and be careful to distinguish between actors with similar names or roles. For instance we will have a number of types of control room user: "operator", "EOIC", "commissioning physicist", "machine development physicist", "Program Deputy" etc. We should construct a list of these and use them consistently.

4.2.4 CONSTRUCTING THE USE CASE

Constructing a use case is an iterative process. The literature warns use case developers not to try to be too precise too early – just get the use case down in outline, especially at the requirements stage. Also don't worry initially about overlapping use cases among different subsystems – once they're outlined we can normalize them later. Also remember at the requirements stage, not to try hard to make them hierarchical.

Each use case is to establish a useful value. When trying to decide on whether some process involves two use cases or one, look at whether one or two items of useful information are wanted by the actor as end results.

These considerations come up everywhere in the literature:

- What tasks does the actor want the system to perform?
- What information must the actor provide the system?
- What information must the system provide the actor?
- Are there events the actor must inform the system about?
- Are there event the system must tell the actor about?
- How does the actor initiate the transaction, or shut down the system?

- Think about the real world objects and their attributes: Do you want to provide a way to “Display all the horizontal BPMs?”; do you want to “Minimize the excitation of all correctors?”

4.2.5 USE CASE DOCUMENTATION

A use case is documented with a combination of text and diagram. The Unified Modeling Language provides a standard for the diagramming, but its importance should not be overplayed. 90% of a use case is a textual list of the process that is gone through to perform the use case. Template documents for writing use cases should be written.

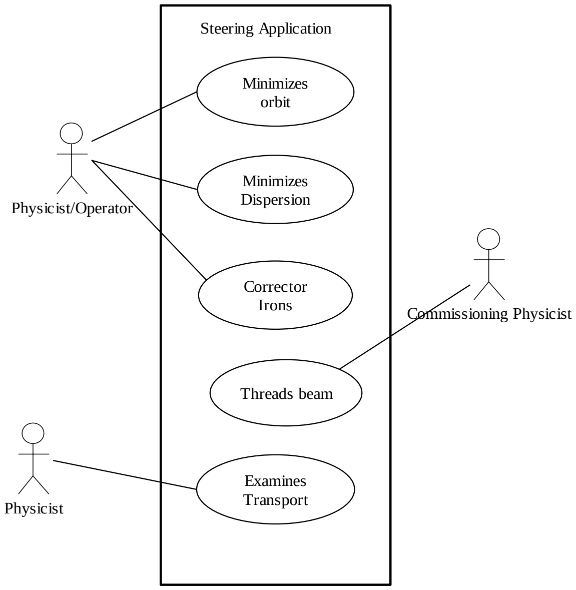


Figure 6: Features level Use Case diagram for the Steering system

4.2.5.1 Use Case details for Needs

For documenting Needs, its sufficient to name each use case, and in fact this stage is options.

4.2.5.2 Use Case details for Features

For documenting Features, each use case should be uniquely named, and elaborated. The elaboration is a short textual description of the intention of the use case, specifying the item of value that the use case results in. A description of the actors’ roles should also be included.

Additionally, a use case “scenario” should describe the flow of events for each particular use case. Still the scenario only documents how the system will be used, not how it works. For instance, the scenario for orbit minimization may be:

The **control system user** prepares by defining which region of the accelerator they want to steer and they make sure the BPM measurement definition can accommodate that setup. They chose whether the system transport matrices to be used in the minimization should be calculated from Twiss parameters or from RMATs. Then they choose a steering "method", scan the BPMs and request that the calculation is made to find a solution for corrector strengths which will minimize the RMS of the orbit. They may plot the scanned and predicted orbits, and the required corrector changes, and if all looks well, implement the solution by trimming or perturbing the correctors to the computed solution values. They may re-scan, and repeat the process. The process is not specifically terminated by any event, and the iteration may be halted at any time.

Figure 7: Scenario for the Orbit Minimization use case

There may be a formal set of guidelines for preparation of scenarios.

4.2.5.3 Use Case details for Requirements

For the purposes of specific requirements, the process of each use case must be described. This is made out as a textual list. Where iteration and branching is involved the logic is described only at the discursive level, labels are not used. The use case outline reads very similarly to a user guide.

Use Case Name: Orbit Minimization

Brief Description: The procedure by which a control system user can calculate corrector magnet settings which will have the effect of minimizing the orbit offset of the beam throughout one section of the accelerator.

Outline flow of events:

1. The control system user begins by following the procedure for setting up steering for a selected region. [This will later be identified as a “<<uses>>” stereotype, see 4.2.5.4]
2. They select a method for steering using the "STEER METHOD" button on the main Steering panel. The default is the SVD, which is recommended for most occasions.
3. If they would like to correct dispersion as well as orbit, they must enter a non-0 "ETA FACTOR". Dispersion correction is only supported by the SVD method.
4. They must then take data from all the BPMs in the region. That's achieved with the "SCAN BPMS " button also on the main steering panel. If there's a non-0 Eta-factor, the dispersion at each BPM will also be measured by this button at the same time. That is an invasive procedure, since it involves momentarily changing the energy of the beam, so if working with colleagues in a ring program it may be worthwhile their warning them before they hit the SCAN BPMS button.

5. It may be desirable to remove some BPMs or correctors from the correction. The control system user should do this using the Setup panel. If steering dispersion, then the measurement of the dispersion at each BPM can also be removed from the fit. That is also controlled through the Setup panel. Additionally, the user can tell steering to consider only some contiguous subset of regions in the beamline using the MICRO RANGE button. If steering the very beginning of the linac, the "SPCIAL RANGE" button can be used to include or exclude the CID. When included it is steered and trimmed before the rest of the region.
6. The default for Dispersion correction will not be to "Steer to the model", so the user should take care when correcting dispersion in X in damping rings. They can remove correctors and BPMs from the correction, which overcomes this problem to some extent, using the setup panel. If desired use the option button which does enable steering to the model.
7. The user then starts the calculation of a minimization solution with the CALC PREDCT TRAJEC button. When this is pressed it will submit the minimization problem to the currently selected steering method. The minimization should only take a few seconds. If its taking too long the user should check the Trouble Shooting use case.
8. Plots can be generated of the "before and after" picture of a solution.
9. It is usual practice for the user to repeat the process of scanning, calculating a solution, and implementing the solution, a number of times. A single button, "SCAN CALC TRIM" can do all three stages in one go. And a further button, "NUM OF ITERA/TIONS" will iterate the process a specified number of times.

Figure 8: Use Case Description for the Orbit Minimization Use Case

The use case description specifies which actors take each action, in this case there was only one actor – a “control system user”. The use case also made reference to one other use case, named Trouble Shooting. The use case also described two sub-procedures which may be converted to uses cases later – the procedure for setting up steering for a region (in item 1) and the procedure for selecting BPMs (in item 5). Note that the process for getting BPM data may involve another actor – the subsystem for acquiring BPM data. At the requirements stage though we would probably not identify that as a separate actor. It may be defined as an actor later, during design.

We will create a template for specifying use cases.

4.2.5.4 Use case Relationships

As the use cases are augmented some hierarchical structure is likely to emerge among the use cases. Two kinds of relationship in particular are supported in UML, the **uses** and **extends** relations, which are oriented towards posing the requirements in terms of Object Oriented Design. UML calls relations of this kind “stereotypes”.

The “uses” stereotype specifies that one use case uses *all* of another use case. For instance, the Minimize Orbit use case may well use another use case for “Plot the BPM orbit”:

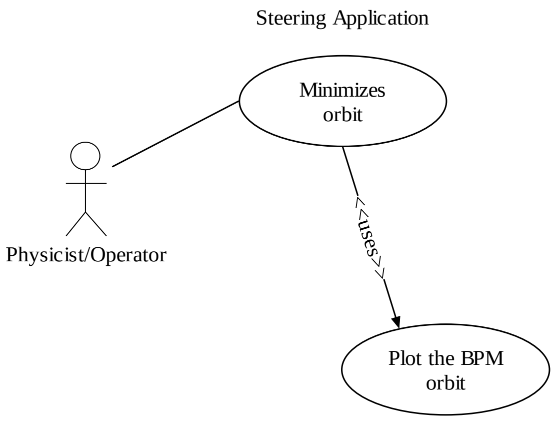


Figure 9: Use Cases with <<uses>> stereotype

The “extends” stereotype indicates a conditional, or exceptional case. For instance, in the SCP the Minimize Dispersion use case has actually been implemented as an extends case of the minimize orbit use case because it is basically an option of minimizing the orbit.

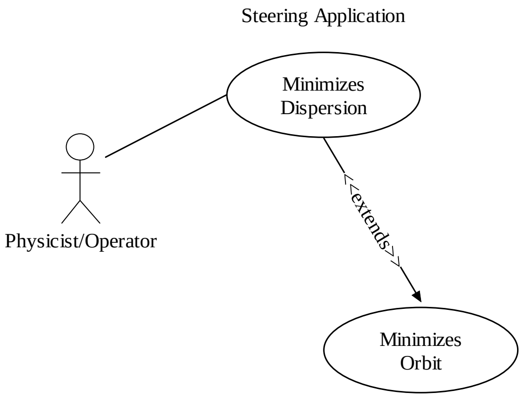


Figure 10: Use case <<extends>> stereotype

Caution: The literature often warns of two things:

- Too extensive a use of the uses and extends stereotypes. Use cases should above all be easy to follow as a guide to the basic intention of the system, especially when defining systems high level requirements.
- The possibility of confusion between extends and uses stereotypes. Apparently some people favor dropping the distinction altogether.

The diagrams become more important as we move toward design because the use cases show how some are specializations of others, extensions of others and so on.

Collections of use cases can be put together to form a **use case package**. This should help verify the demarcation, or “treaty points” between subsystems. Frequently this is oriented towards collecting packages that deal with the same actor or activity. Also some use cases become clearly primary, some secondary and some optional. This classification process forms the basis of identifying the class collaborations in an Object Oriented design.

4.3 SUPPLEMENTARY REQUIREMENTS

The deliverable artifacts for the Supplementary specifications are given in Table 4: Deliverables for Each Subsystem. Here we describe these in a little more detail.

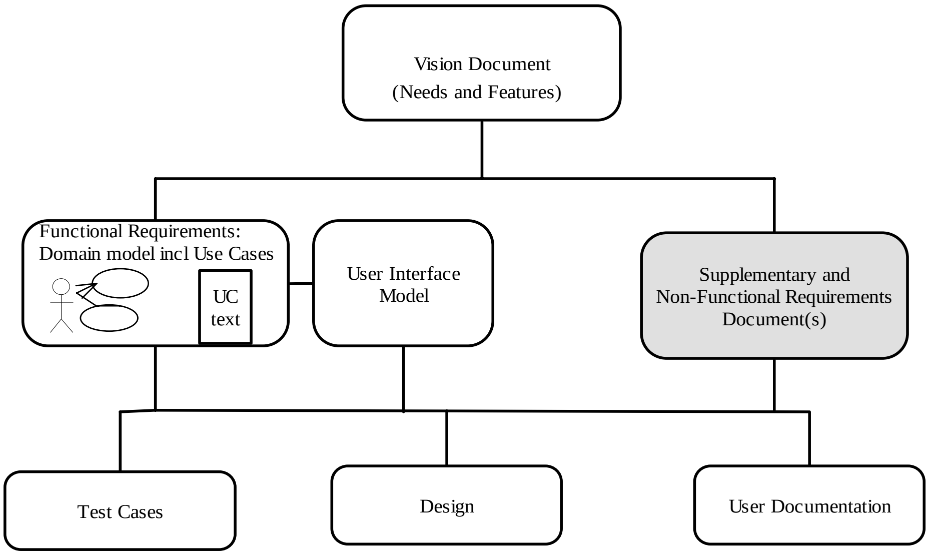


Figure 11: Where the Supplementary requirements fit in to the core deliverable hierarchy

4.3.1 NON-FUNCTIONAL REQUIREMENTS

The non-functional requirements specify those aspects of the system which are not related to its desired behaviour (which is specified by its functional requirements). Non-Functional Requirements most importantly include:

- Performance requirements (a summary for this system from the Sizing and Performance Specification described in 5.1).
- System requirements. This gives constraints on the technology systems which must be used, e.g. TCP/IP, VME, EPICS etc.
- Architectural Specification. This section specifies the software architectural model constraints. For instance, that an application must employ a WWW browser client, or that it must be 3-tier, or that its client-server with its user interface must on NT and its server on Unix. This is particularly relevant at the Elaboration stage.
- Design recommendations. This section keeps track of ideas regarding the implementation, while the requirements are still being pursued.
- Forward References. This section keeps track of ideas, constraints, or requirements which have been discovered but are not yet ready to be pursued. Things we have to bear in mind for the future.
- Cross-reference constraints. The constraints imposed by other systems, including references to their artifacts

- References. Bibliography and such.

A template for the Non-Functional specifications will be provided.

4.3.2 CONFLICT MATRIX

This shall define the needs of each control subsystem (RF, feedback, MPS) regarding its priority in contention situations with other systems. This data will be used for constructing the Contention Handler system.

4.3.3 TAXONOMY

This is a list of the key systematic characteristics of the system. The data requirements, such as pulsed data requirement, synchronization, latency, etc are described in the Sizing and Performance Specification (see 5.1).

Property	**Description**	**Example or List of Valid Entries**
Initiation and Termination	How is the system started and terminated.	Manually or Batch or Job
Duration	For how long is the system typically active and not idle	Eg: For steering: between 10mins & 4 hours For acquisition system: continuous
Frequency	How often is it initiated.	Continuous or As Needed or Every n seconds
Read/write	Is the process mainly a reader or writer	Read or Read-Write or Write
Running State	What machine state is this system for?	“Normal running” or “Startup only” or “all”.
Shareable input	Is the pulse related input data of this system shared by other systems.	Eg. For feedback this is yes, because BPM input data is shared.

Figure 12: Subsystem Taxonomy

4.3.4 REQUIREMENTS AND TRACEABILITY MATRICES

The needs, features and consequent requirements, plus supporting data, can be entered into RequistePro as a relational database. From this, the hierarchy between needs, features, and requirements, and the relationship between the requirements themselves can be constructed (see Figure 2). Additionally, some important “attributes” of each requirement can also be managed, such as its status, cost, priority, etc (see 6.1.2 RequisitePro Requirement Attribute Tags).

4.3.5 COST BENEFIT ANALYSIS

For some systems there is likely to be a full analysis of the benefits of various alternative implementations (see 5.3.2). There may also be a quantitative cost-benefit analysis of each system based on some business model, in addition to the cost benefit analysis as a whole (see 5.3)

5 REQUIREMENTS AND ANALYSIS MANAGEMENT AND ARCHITECTURE

This section describes the requirements analysis for the control system as a whole. This includes the identification of the constituent systems and how we would describe their collaboration. It also touches on how this overall planning is communicated to the NLC executive. Table 6 itemizes the core deliverables of the Requirements and Analysis effort for this overall work. Items in gray are not described further.

Table 6: Deliverables for Management and Architecture

Artifact	**Description**	**Authors / Ownership**	**Mainly When**	**Contents**	**Audience**
CDR summary	Summary of requirements work breakdown	Management, Architect	Inception	As Spence Specified.	Reviewers, Executive, User group, Champions
WBS	Work Breakdown Structure	Management	Inception	GANT chart of activities	Executive, Management
Basic Design Principles	Statement of Intent for core systems, and their interoperability	Management, Architects, Champions, Designers	Inception	Summaries of basic architecture, utilized technologies, outline design of main systems	Everyone
Developers Guide	Combined BUG & POOP	Designers, Developers	Inception, Elaboration and Construction	Programmers Guides to BPM, RF, Timing Magnet, Analog & digital status etc. APIs.	Designers, Developers
Sizing and Performance Specification	Diagrams and text showing nodes, data flow, and timing requirement, for each major data source (BPM, RF, feedback etc)	Network Architect, Infrastructure Champion, Analysts	Inception	Breakdown of data types by functional requirement and performance requirement (real-time, soft real-time, digital, control etc)	User group, Champions, Architects, Analysts, Designers, to verify “streaming”, archiving and delivery requirements.
System Boundary Analysis	Summary of System Boundaries and Treaty Points	Managers, Architects, Designers	Inception	A single point of reference for system boundary info	Architects, Designers, Developers, Champions

Artifact	Description	Authors / Ownership	Mainly When	Contents	Audience
Domain Analysis	Document and Diagrams detailing domain objects and their relationship	Architect, Analysts, Champions, Designers	Inception and Elaboration	Text and diagrams describing hardware devices, drivers, filed buses, controllers, networks, data infrastructure, db, applications and their relationships.	Use Case Specifiers, Analysts, Designers
Glossary	A dictionary of terms.	Everyone	Inception and Elaboration	Definitions together with references. Cross referenced to WBS.	Everyone
References	Pointers to literature, both accl. Physics and Engineering and programming	Everyone	Inception and Elaboration	An organized set of the basic literature about each sub-system and overall systems.	Everyone, so all stakeholders can learn enough to understand requirements.

5.1 BASIC DESIGN PRINCIPLES

This should be done in two phases: an early goal of the Inception phase should be to produce a statement of the core design *proposals* as a distinct reference document. Later this document will be revised to say specifically which overall design and systems we are going to use. This would be a major work stemming primarily from the Architecture R&D work we have ongoing. For the purposes of helping to satisfy requirements and analysis goals it would include:

- A summary of the absolutely core needs for the control and acquisition for the accelerator.
- What are the core systems, and which are secondary or built on top of these? This might include a Pareto diagram from the core needs to the needs of each subsystem
- Data infrastructure, including middleware “layers”; the plan for the streaming and other data paths
- The architecture of the core APIs, and examples. For instance, are APIs to be service oriented, like CDEV, or framework oriented, like MFC.
- Application architecture. Are we going to standardize on an application architecture like three-tier, or client server, or host based (like SLC)? Are we going to use an Application

server, if so which one? What will be the plan for multi-platform development? To what extent are we shooting for or enforcing Object Oriented design?

- Enterprise database schema in outline, including plan for non-device type data.
- Technologies and computational 3rd party systems that it is known early on that we are going to use: For example
  - At the low level – which busses etc: Infiniband, VXI etc.
  - At the high level – is Windows always going to be the user client?

Probably, the proposal phase of this work should be started immediately after the Needs of each system are known from the Vision documents, and the Sizing and Performance characteristics have been analyzed. One item to include would be a clear statement of intent about the role of EPICS; is it a system from which we a are going to take and develop appropriate technology, or is it a system which we are going to develop into something which can control an accelerator like the NLC?

5.2 DATA SIZING AND PERFORMANCE SPECIFICATION

The network sizing requirement will be tackled both from the source device level and application levels. By "device" level, is meant the requirement to push or take control data by each control subsystem, such as BPM, RF, MPS and so on. By "application" level is meant the functional requirement for data of interactive applications, and the requirements of offline data analysis programs. As far as possible we'll devise a taxonomy for describing the data requirement for all of these, so that they can be accumulated and compared, giving us an accurate picture of the network architecture requirements around the machine, and the requirements for storage and retrieval. The taxonomy will specify units to be used in each category where possible. Each subsystem's champion shall be responsible for specifying these requirements according to the taxonomy. We shall keep the application data sizing data together with the device data so as to keep all the data sizing information in one place.

5.2.1 DATA SOURCE DEVICE SIZING AND PERFORMANCE

The objective for this will be to construct a data size and network bandwidth roll-up for each subsystem of the machine (infectors, rings, main linacs etc) with respect to each control system (such as RF, BPM) and each data type managed by that control subsystem (for instance, for RF, two data types are "RF bunch current vector" and "RF reflected waveform").

For each data type, we hope to create a spreadsheet of data size requirements with columns given in Table 7. Since for each data type, (e.g., for the BPM control subsystem, the "BPM stripline data") is likely to be slightly different at different locations (e.g. QBPMs as opposed to main linac BPMs) we should include a sub-type tag to allow the specifier to identify a location and count of these data items. This will be further specified, but here is an example taxonomy:

Table 7: Data Source Sizing Taxonomy

Control System	Name of device subsystem, such as BPM, RF, Digital Status
Machine System	Name of accelerator sub-system, eg: "e+ Pre-linac Section" or "e+ Main Linac Section".
Data Item Property	**Description**

Property	
Descriptive Name	Data Item's description, E.g "BPM stripline data".
Sub-type	To distinguish data items which describe the same kind of data, but have different size properties. E.g "QBPMs"
Input/Output	Is this item input to, or output from, the device
From Location	From where in the accelerator? A schematic of the NLC network node distribution will be given, and the area names should be listed, e.g. "e+ MDR" or "BC2".
To Location	To where in the control system. If Output data, and is intended for the application level control system, say "Central Host" (?).
Number	Number of devices per location. If different at different locations, use data sheet with different "Sub-type" or "From Location".
Structure	The data's structure: If array what dimensions? If a record, give major fields with sizes or an actual data structure definition.
Volume	Data Quantity per transmission, in bits for entire structure described.
Frequency	In transmissions per second peak, including possible machine development.

5.2.2 APPLICATION LEVEL DATA REQUIREMENT SPECIFICATION

For each application we will summarize the data requirements according to the following taxonomy:

Table 8: Application Level Data Requirement Sizing

Application Name	The name of the user level application
Data Item Property	**Description**
Descriptive Name	Data Item's description, E.g "BPM stripline data".
Sub-type	To distinguish data items which describe the same kind of data, but have different size properties. E.g "QBPMs", or if all sub-types are treated equally say "All".
Use Case(s)	The names of the use cases of the application in which this data is used.
Input/Output	Is this item input to, or output from, the application.

From Locations	From where in the accelerator. If this is an application level data item, say “Central Host”.
To Locations	To where in the accelerator? A schematic of the NLC network node distribution will be given, and the area named should be listed (e.g. “e+ MDR”) or “Everywhere”.
Structure	The data’s structure: If array what dimensions? If a record, give major fields with sizes.
Volume	Data Quantity per transmission, in bits for entire structure described.
Frequency	In transmissions per second peak, including possible machine development.
Latency	Permissible time between request and delivery, given in pulses or in seconds.
Resilience	Tolerance to missing data, given as descriptive text
Timing Specificity	Is there a timing constraint? Is this “pulse” related?
Collective	How would the data be aggregated. For instance, for the magnet control application, is it desirable to collect the data by IOC?

Questions: Which data sources should we do this for? Obviously BPMs, RF, feedback, magnets, digital status – what else? Which applications?

5.3 COST BENEFIT ESTIMATION

In requirements analysis we are trying to help the customer see their cost/benefit function, and decide where on it they want to shoot for. Typically the more functionality the more the project will cost obviously, but that function may well not be linear.

5.3.1 HIERARCHY OF INTENTION

In a project of this size and longevity we will also need to manage the alternative packages of requirements for each system. In contrast to most requirements specifications, we may well need to pursue a number of these alternatives within the requirements and analysis phases, and document their cost/benefit value before selecting one with which to move forward. So we should have a well structured way of tracking alternative scenarios which clearly distinguishes their constituent features. To do that we can use an attribute type named Benefit for each *feature*, whose values are Must, Should, Could, Might, or alternatively “Necessary, High, Medium, Low”.

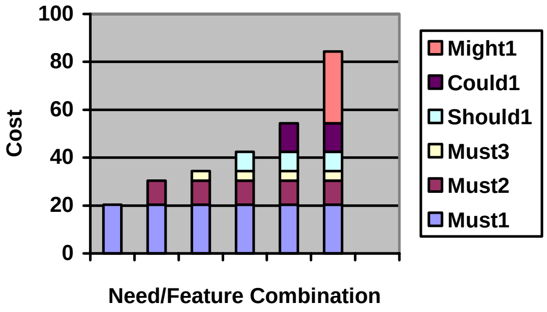
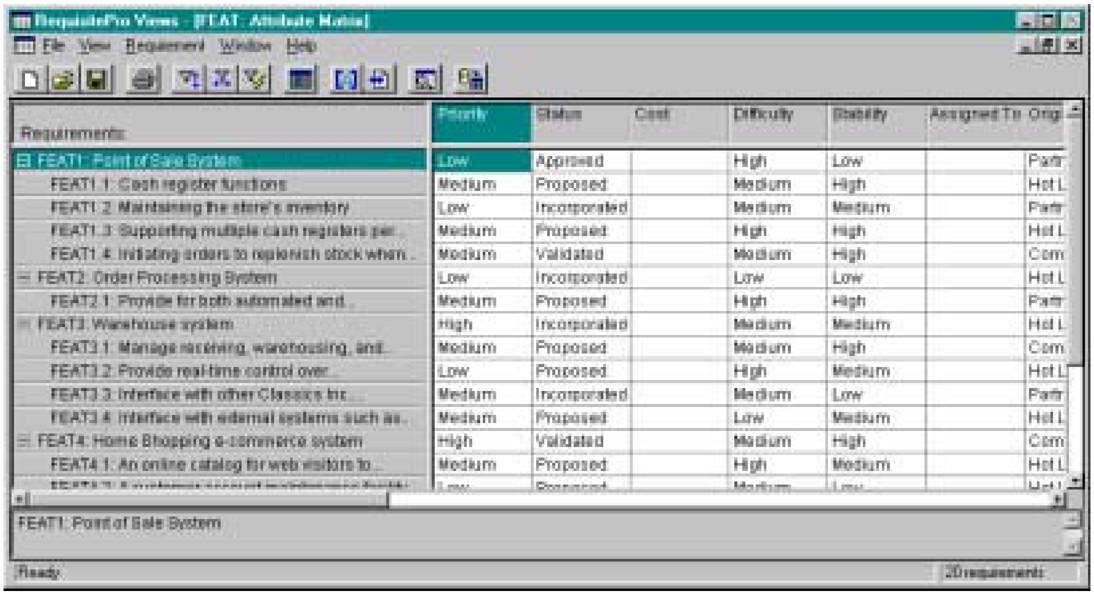


Figure 13: Cost benefit of alternative Feature combinations The features named Must1Must2 and Must3 have to be implemented in any alternative system design, but there are four alternatives given that constraint.

This benefit of a feature will be tracked in RequistePro using a new tag we'll call "Benefit", valued Necessary, High, Medium, Low, Wish. The "Benefit" tag is different to the Priority tag in that the Priority tag is an ongoing indication of which requirement should be tackled at any one time, and is probably at least somewhat dynamic as conditions change. The Benefit tag on the other hand shows the extent to which a feature extends the systems functionality[1]. It's set during the Inception and Elaboration phases, and may be used to:

1. Decide which features, or feature sets to tackle next during the development cycle
2. Help distinguish alternative requirement sets (see 5.3.3 below)

[1] If the dynamic aspects of project management are done through another tool, then it might make sense to use the word "Priority" for the effective benefit of a feature in the requirements specifications and RequisitePro db.



Requirements	Priority	Status	Cost	Difficulty	Stability	Assigned To	Orig
FEAT1: Point of Sale System	Low	Approved		High	Low		Partr
FEAT1.1: Cash register functions	Medium	Proposed		Medium	High		Hot L
FEAT1.2: Maintaining the store's inventory	Low	Incorporated		Medium	Medium		Partr
FEAT1.3: Supporting multiple cash registers per...	Medium	Proposed		High	High		Hot L
FEAT1.4: Initiating orders to replenish stock when...	Medium	Validated		Medium	High		Com
FEAT2: Order Processing System	Low	Incorporated		Low	Low		Hot L
FEAT2.1: Provide for both automated and...	Medium	Proposed		High	High		Partr
FEAT3: Warehouse system	High	Incorporated		Medium	Medium		Hot L
FEAT3.1: Manage receiving, warehousing, and...	Medium	Proposed		Medium	High		Com
FEAT3.2: Provide real-time control over...	Low	Proposed		High	Medium		Hot L
FEAT3.3: Interface with other Classics Inc...	Medium	Incorporated		Medium	Low		Partr
FEAT3.4: Interface with external systems such as...	Medium	Proposed		Low	Medium		Hot L
FEAT4: Home Shopping e-commerce system	High	Validated		Medium	High		Com
FEAT4.1: An online catalog for web visitors to...	Medium	Proposed		High	Medium		Hot L

FEAT1: Point of Sale System

Figure 14: RequistePro Attribute Matrix showing Priority, Status and other attributes of some project FEATures.

The vision document should detail the "Benefit" of different Needs and Features. Those needs and features should be examined in the Cost Benefit Analysis (Table 1: Summary of Deliverables for each), and the costs entered into the RequisitePro requirement db, using the Cost attribute of each feature (see above), so that the Work Breakdown Structure (WBS) can be accumulated (unless of course other software exists for doing this better). The vision document should reference the cost benefit work.

5.3.2 RISK

To help manage scope creep, and feed into the project planning process, we should explicitly identify requirements which have the potential to negatively impact schedule and budget. That can be folded into the RequisitePro db too: a "Risk" attribute could be assigned to each feature, valued simply "high", "medium" or "low", where a high-risk feature would be one which has, for instance, the potential to overrun badly. The risk analysis can then use that information directly, and project planning can then use the Effort, Benefit, and Risk attributes combined to schedule work and manage scope (see Table 10: RequisitePro Attribute Tags Proposed). Of course we may be using a separate tool for project management, in which case we'll have to interface that to RequistePro somehow.

5.3.3 ALTERNATIVE REQUIREMENT SETS

We have all been in the position where there are a number of alternative ways of achieving the same need. Often one alternative is elegant but would take more time, while another is quick-and-dirty, and there may be some combination of these. We would like to detail the actual requirements necessary to achieve these alternatives to cost them out:

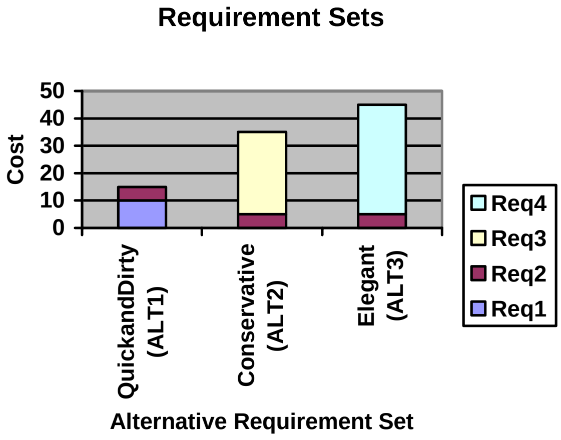


Figure 15: Three alternative requirement sets. Only requirement2 is common to all three alternatives.

Figure 15 shows three alternative requirement sets. These should be identified as alternatives in the Vision document. When described in the vision document, use cases and other documents, the individual features or requirements making up these alternatives can be identified by a RequisitePro requirement type which we can name ALT, for "alternative" (see Table 9). In the example above, Req2 will be linked to ALT1, ALT2 and ALT3 to show that its part of all three alternatives. RequisitePro's "Traceability matrix" (see Figure 2) can then be used to manage alternative requirement sets.

Additionally, some alternatives will yield in fact more benefit than others, so an ALTernative may well be described by a Benefit attribute tag (see 5.3.1).

5.3.4 ALTERNATIVE REQUIREMENTS FOR COLLABORATING SOFTWARE

In a functionally decomposed system like ours the functionality of one subsystem greatly affects the features needed in associated subsystems. Clearly there may well be different combinations of requirements which achieve the same needs, i.e. if one system does one thing then another doesn't have to. For example, if the "archiver" archives error messages, and the archive browser allows us to browse text messages, then the effort and cost of the "error display browser" goes down markedly because it is just an instance of an archive browser.

This can be examined and managed by cross-referencing alternative requirement sets (ALTs) across sub-system requirements documents. RequisitePro includes the facility for cross-referencing in this way – a traceability matrix can be constructed across projects. We can use that to help find the minimum cost combinations.

5.4 MANAGING SCOPE AND CHANGE

Scope creep is obviously most damaging when its impact is realized late, or in a foundation component of the system. So scope creep is minimized by this formal approach in a few ways:

- Early and explicit identification of the high and low level use cases
- Traceability, and requirements matrices allows us to see the impact of scope changes
- Clearly identifying the system specification level requirements that result from needs and features, so we can see early how much programming work in fact each feature entails
- Explicit estimation of the “Risk” of overrun of each feature to the overall project, together with its Effort and Benefit
- Clear identification of the core principles and systems of the control system, and keeping each system small and clean. By the avoidance of stove-piping
- Explicit and honest expression of the alternatives, their benefit, and their impact on the requirements of related systems, is expressed early, and clearly.

In a project with as long a timeline for the development as the NLC, it’ll be difficult to assess the impact of change requests on our baseline. It’s hoped that the iterative structure of this methodology, with a commitment to hard deadlines in each iteration, will ameliorate that risk.

Clearly we should additionally put in place a formal process for accepting and accessing change requests.

5.5 EDUCATION

We can prepare for the requirements and analysis effort through three sources of education, regular classes, participation in ongoing B-factory activities, and analysis of the existing control system. In addition to these, we should create a new-hire package to help orient new team members.

5.5.1 CLASSES

We should provide for the education of the development team in the core science and technology, just as was done for the SLC. This includes; basics of the accelerator and the physics problem, accelerator physics instrumentation and control, and commissioning problems that will be addressed by software. Obviously these classes are intended for people in the development team who are not familiar with the subject already. Other classes in the technology we intended to use should be scheduled, for instance in Java and related topics. On a personal note I think we would benefit from more participation in software and applied computing conferences.

5.5.2 CONTROL ROOM PARTICIPATION

This is a particularly important education for new members of the development team. In order to appreciate the peculiar needs of accelerator commissioning and operation its necessary to experience it for oneself. While we have the benefit of an on-site accelerator we should encourage the participation of developers in control room activities so that they’re exposed to more systems. For instance we might rotate a “Developer in Operations” one day a month each.

5.5.3 ANALYSIS OF THE EXISTING CONTROL SYSTEM

We should interview the principal designers of the existing control system and examine the existing control system to evaluate what was done right, so we can do it right again.

We should examine the existing source code, to see what proportion of it does what jobs. Then we can concentrate effort in providing development tools and services to cut down programming time consumers. In the SLC these included message issuance and error handling, and panel handling. A particularly effective time saving service was Handypak. The message service was defined very early, and virtual micros provided a test bed before the real things were in place.

5.6 TOOLS

This section introduces some tools for requirements analysis and design which we may want to bring in to our development environment. This set should be kept small and cheap if we hope to use other labs in collaboration. Rational RequistePro has been discussed in detail elsewhere.

5.6.1 RATIONAL ROSE

Rose is an expensive CASE tool for UML diagramming, which we would use for design. It interfaces to RequistePro and provides for analysis of real-time data flows as well as the regular object oriented analysis and design visualization tools. We should be careful not to over emphasize diagramming for object oriented design though, the literature warns of an analysis paralysis that can consume projects designed this way.

5.6.2 BASIC DIAGRAMMING TOOLS

Smartdraw is a popular shareware application that includes a comprehensive library of UML diagramming objects (the graphics in this document were created with it). It’s available at http://www.smartdraw.com/

Visio is Microsoft’s diagramming tool. SLAC has a site license for it, but the run-time licenses are restricted.

.

6 APPENDICES

6.1 REQUISITEPRO CONFIGURATION

6.1.1 REQUISITEPRO "REQUIREMENT TYPES"

In RequisitePro a "requirement" is defined very broadly. This is a proposed list of the kinds of items whose hierarchy and cross-references we can track using RequisitePro

Table 9: "Requirement Types" tracked by RequisitePro

RequistePro Abbrev.	**Type**	**Description**
PURP	Purpose	The goal of a subsystem or component set.
NEED	Need	A top-level functional requirement., such as might be advertised on the back of a shrink-wrapped software box.
FEAT	Feature	A mid-level functional requirement, such as might be advertised in the blurb for a version release.
UC	Use Case	A description of a use of the system, described from an actors point of view.
REQ	Requirement	A specific item of programmed behaviour, of which many are necessary to satisfy a use case
ALT	Alternative	An alternative NEED, or FEAT, and REQ set which can satisfy the purpose.
TP	Treaty Point	An item from the specification of the system boundary.
DBREL	Database Relation	A required relationship between DBNAMES in a db, 1:1, m: 1 or m:n
DBNAME	Database entity	A database element, being a collection of attributes. This may describe a device, or some other aggregate.
TESTC	Test Case	A test suite, aimed at testing a Feature.
TESTI	Test item	A specific test in a test case.
GLOSS	Glossary term	A definition of a technical term.

6.1.2 REQUISITEPRO REQUIREMENT ATTRIBUTE TAGS

This is a proposed list of the RequisitePro "Requirement attributes" that we could use. All except Benefit and Risk are shipped with RequisitePro. We should decide which of these attributes to attach to each one of the Requirement Types listed above.

Table 10: RequisitePro Attribute Tags Proposed

Attribute Tag	**Description**
Benefit	Describes the desirability of a need, feature or requirement.
Effort	Estimation of the effort necessary, units to be decided
Risk	The extent to which a feature or requirement has the potential to impact schedule or other costs.
Cost	Extraordinary capital costs, such as necessary hardware. Need more guidelines for this
Priority	The ongoing priority assigned to a feature or need – updated during the development
Status	The stage of approval of a Need or Feature. Valued "Proposed", "Approved", "Elaborated", "Designed", "Implemented"
Difficulty	How hard is it to implement this Need, Feature or Requirement
Stability	How likely is the feature or requirement to change in the future
Assigned to:	An individual
Originator:	An individual

Another possible tag is CP, whether feature or requirement is on the critical path?

6.2 DOCUMENT TEMPLATES

(Not yet provided)

6.3 QUESTIONS

1) How long should the development cycle be? 10 weeks?

2) How do we keep the WBS and requirements documents in concert?

6.4 BIBLIOGRAPHY

Bertalanffy, Ludwig von, Brazziler, George. General System Theory, 1968
Blanchard B. System Engineering Management, Willey 1998
Graham, Ian. Requirements Engineering and Rapid Development, Addison-Wesley 1998
Liberty, Jesse. Object Oriented Analysis and Design, , Wrox Press 1998
Mumford, Enid. Designing Systems for Business Success, Manchester Business School, 1986
Rational. Requirements management with Use Cases, 1998
Wiegers, Karl E. Software Requirements, Microsoft Press 1999

6.5 IMPLEMENTATION PLAN

The following is an incomplete outline of the steps to implementing this methodology:

1. Approve methodology in general, and create template deliverables.
2. List and buy-off deliverables for each job type in Table 3: Job Descriptions, to get to the stage where its clear for which deliverables each job function is responsible. This may include composing workflow diagrams(?)
3. Send memo to NLC executive detailing our proposed changes, and the role of our users, such as project Champions, in this scheme.
4. Get buy-in from Champions, making clear their role and deliverables in this methodology.
5. Agree and finalize common taxonomies with the system champions:
    a. Data Sizing and Performance.
    b. Conflict Matrix. Identify key information and data which will be used to resolve sub-system conflicts
6. Install RequistePro and configure for requirement types and attributes decided on above.
7. Install and standardize on a diagramming tool (Smartdraw, Visio or another)
8. Create a development environment for requirements (i.e. Clearcase VOBs for Requirements, web space, and assign web access privileges).
9. Create a development procedure for requirements and analysis from this document. This will include such things as the procedure for assessing each development iteration, the requirements review process, and others.
10. Approve and schedule classes and the rotation of developers in the control room.
11. Capture a common vocabulary in a Glossary and publish it in a place where everyone has read-write access.
12. Agree a list of actors.
13. Decide on the overall workflow cycle: 10 weeks, 5 weeks?
14. Create a requirements and analysis practices committee to review the effectiveness of this procedure and make improvements at fixed intervals, such as quarterly.