Guidelines for Software Development

Last Date : May 7, 2001 Release 2.7

Solutions Engineering Corporation

Purpose.

The purpose of these Guidelines is to the improve the quality of products we provide our customers/clients and to decrease the costs of these products to our clients. The purpose then is to focus on three subjects – 1)product, 2)quality, 3)cost. Everything in today's literature regarding these three subjects is centered around "Process". Vanguard among these is "Six Sigma" (Motorola and GE's mantra for successful improvement in product quality and profitability), CMM (Capability Maturity Model advanced by the Software Institute), the Unified Software Development Process (Jacobson, Booch, Rumbagh), and RAD (Rapid Application Development). Nothing in the Guidelines is particularly unique to this text. Everything which appears here could be found in one or more current "Best Practices" books or articles in the field of application software development. We have simply distilled these down to the essentials relevant to the development of client-focused application software. We have also combined these with the lessons learned in more than 20 years of experience engaged in this work.

Software products are not just executable code. The **product** includes :

- executable program(s),
- installation process,
- user documentation,
- system and installation configuration documentation,
- support documentation,
- collective knowledge regarding the use, maintenance, and properties of the product.

Depending on the needs of the client, the product may also include :

- data migration,
- user help desk,
- version control,
- sequential or over-lapping version planning,
- multiple platform support,
- integration services,
- and others.

The **quality** of our products is defined in several terms :

- Full Functionality in relation to the complete and prioritized set of needs as defined by the client,
- Automation - Workflow of the system in relation to human process and time – minimizing both,
- Usability - Time required for users to gain operational proficiency and confidence, often referred to as "user friendliness".

- Stability - Error free operation and use, regarding execution path and database states,
- Vulnerability - Time and effort required to recover from system disaster,
- Extensibility – ease with which the product can be modified to accommodate evolving client needs - workflow models, analysis requirements, systems interfaces, user loads, data volumes, etc.

The **costs** of products are the result of the complex combination of resources required to produce the required product/quality. These resources include :

- Personnel (in-house and contract)/expertise hours,
- software/hardware tools,
- systems configurations,
- existing components,
- risks and risk control techniques,
- calendar time,

The most costly of these is usually the personnel/expertise – the people with specific expertise required to produce the target product/quality. In fact, the cost and strategies for mitigating the costs of the other resources revolves around how it will impact the costs of personnel/expertise. Every project is filled with lots of decisions – what to do, what not to do. Every project consumes additional resources because actions were taken which did not improve the product measurably. If an action does not improve, or has a low likelihood of improving, product functionality and quality (in terms valuable to the client) then it should not be done. These actions delay projects and confuse and distract the team members.

We aspire to improve product quality and project efficiency through these Guidelines. Every project definition will have the following sections or elements :

- Background
- Requirements
- Risks
- Design
- Development
- Test
- Document
- Support
- Enhance

All of these sections or elements are completed in a timely, complete, and logical manner through careful planning and execution. This is accomplished through the continual monitoring and management of a program manager. This is an often overlooked resource required to maximize the efficiency of all project resources.

Background.

The Background provides the environment of the client, through the perspective of the client. It defines in very brief terms the domain of the client – their business model, their operational properties. It provides the context in which the project or system is developed. The Background defines an overview of the customer problem domain, providing :

- business model of the client,
- operations and workflows of the client,
- an overview of the problem/project,
- scale and scope of the project,
- number of users or personnel affected,
- current workflow and systems being replaced or enhanced,
- types of users and stakeholders and critical issues regarding these,
- factors affecting value or importance of the system.

This might include the vision and mission of the customer, the expectations of the customer should be identified. Sections of the customer web site or published material may be included or referenced here.

Requirements.

This section should include all of the requirements of the system. If a requirements management system is already in use, then the Project reference should be provided and the output text file from the system should be included in this section, or be in an Appendix referenced in the requirements section. Regardless of how the requirements are being recorded, each should have the following attributes :

- Name – short unique reference,
- Description – brief, concise description,
- Status – indicates status at the particular time (Proposed, Approved, Disapproved, Design-valid, Tested),
- Priority – indicates importance to Project (Critical, Important, Ancillary),
- Estimated Cost - (personnel time – many times this awaits Use Case definition),
- Risk level - associated with the requirement (Critical, Significant, Ordinary),
- Category – groups it to a logical grouping for the Project (see below),
- Author(s) – persons identifying the requirement as such,
- DateCreated – the date this requirement was created,
- DateModified – the date this requirement was modified (change of any value),
- History – may be required for larger, more complex systems.

This section must be comprehensive and indicate the sequence in which the requirements should be met (its target release number and its priority). All requirements should be grouped according to subject matter. Mandatory subjects include:

1) Organizational Requirements.
   a) Objectives for the system or project. What are the primary objectives to be attained in the construction and implementation of the system? What are their priorities? What are the measures of success for the project? How important is each to the overall measure of success? These should provide a very clear definition of the goals of the system – the metric for project success or failure.
   b) Constraints. What are the constraints which must be met? Are there budgetary constraints, time constraints, personnel time commitment constraints? Do the objectives need to be met in stages? Will it improve the overall ROI (Return on Investment) if the system is deployed in stages or will it result in too great a disruption to the operations and staff training and transition? Constraints define the limits of features and capabilities. There has never been a real system constructed where costs and time to deploy were not limited. All too often the question "Can you do feature_Z?" is answered strictly on a technical basis, ignoring the budgetary and time constraints. These are as real as any technical limitations and should be treated with the same respect.

2) Functional Requirements
   a) Workflow (Areas identified for each functional operation of the customer problem domain, the support of the workflow must be complete within economies and provide for the completion of each required workflow), Allowing user selectable variability in the workflow, or strictly enforcing specific workflows, defining allowed variability through setup operations.
   b) Automation (Objectives in workflow automation; requirements and constraints on these operations),
   c) Human interface requirements, error trapping requirements (stringent or relaxed), the expertise level of the user community, the areas where data quality is critical and where it is less important, Section 508 compliance,
   d) Business Rule Enforcement – rules which affect object properties and behavior, possible states, possible state changes, the range of rules, their classification, how they will be defined (maintained) and how they will act,
   e) Report Writing/Analyses needed in order to support management or business decisions. This includes the fixed or scheduled Reporting, *ad hoc* Reporting, Charts, Line Graphs, Bar Charts, Pie Charts, Tables, Summaries, Statistical Analyses, data selection/manipulation/export.
   f) Systems Interfaces needed to inter-operate with other systems needed by the client. These might include Accounting systems, Data Collection systems, Personnel systems, Asset Management systems, Security Systems, or others.

3) Non-Functional Requirements
   a) Target Workstation Environment,
   b) Target Server & Infrastructure Environment,
   c) Network architecture, bandwidth, and shared load environments,
   d) Time/Work/Personnel properties,

e) Training requirements,
f) Acceptable support resource requirements,
g) Security,
h) Portability, Extensibility, Enhanceability, Supportability,
i) Other.

4) Non-Requirements – if there are any requirements which are intuitive or reasonable to the analyst in the context of their understanding of the target environment, but are declared non-requirements by the stakeholders, it is wise to record these explicitly. Building systems to meet non-requirements is un-necessarily expensive, time consuming, and complex. This is where wish-lists are pruned to a list of intelligent investments.

Risks.

Each and every known risk for the system should be identified. A risk is anything which could hinder the system from meeting the stated requirements or jeopardize the feasibility of achieving certain goals in a project. This includes meeting operational, timeline, and/or budgetary constraints. Virtually every software, database, or data warehouse project which fails or is terminated is the result of failure to identify risks and their importance at the time they were knowable. In most cases the plug is pulled after the project consumes substantial resources. No one likes to hear about risks. It is important to keep team spirits high and the energy focused. But risks are real and have very real consequences to project viability and success. The irony is that risks are much like any other project issue or property – the closer you embrace the matter, the clearer the solution becomes and, if dealt with properly, risks become tasks. They get identified and characterized, and then they are overcome. Some risks however dramatically change the resources required to meet project goals. This is time for management to modify the goals/requirements, change resource allocation, or sometimes scrub the project. No one likes to scrub projects. Team members invest some part of ourselves (our lives) in a project and much of our reward is seeing the result and its affect on the client organization. But it is far better to scrub a project earlier than later – saving money, time, and opportunity. All risks encountered in the process of Requirements Definition and Data Source Definition will be described in “risk” sections of their respective documents, which comprise the project deliverables.

Potential risks include :

1) Requirements related – are the requirements complete and properly understood,
2) Requirements in the context of the constraints – is it feasible to meet all of the requirements and the defined constraints?
3) Technical issues – can the required system be constructed to operate within required parameters on the target systems with the defined tool set?,
4) Unconfirmed technologies or tools applied in a new manner,
5) System components which may present issues when operating in the target environment (e.g., will the database operations perform correctly over the designated bandwidth?),

6) People resources, experience, expertise, continuity, (is all the necessary talent onboard in each area of the project - project management, systems analysts, design, database engineering, programming, testing, configuration engineering, documentation)
7) Funding and Time availability,
8) Stakeholder expertise, consensus, and availability (e.g., will client personnel be available to resolve questions as they arise? Will they agree on issue resolutions? Do they have the expertise to provide informed, correct, and complete guidance on problem domain issues? Can they articulate the requirements for the project?),
9) Ambiguities of business rules or lack of resolution among stakeholders regarding business rules,
10) Inherent subjectivity of some subset of business rules.

Design.

The system design is comprised of all of the material which communicate how the target system is to behave. All systems produced under direction of these guidelines will be "Use Case Driven". The design will be composed almost exclusively of the UML components required to clearly, accurately, and comprehensively define the target system. Every system detail which is uniform across all systems developed by our teams, is covered in "Systems Engineering Guidelines". These guidelines should be used throughout all systems unless countermanded by the design of a specific system.
The Data Dictionary and Class Diagrams of the system will be defined according to the needs of the system as required by the business rules of the system domain, the reporting requirements as defined by the stakeholders, and cooperating subsystems in the properties and requirements of related systems. The developers are at liberty to add to the dictionary if needed in the course of implementation (to enhance performance, improve human automation, enforce data consistency, or meet the needs of other issues). In most cases The Class Diagrams (data dictionary) have undergone considerable review and examination. Therefore changes are not recommended to be made in development without consultation with the design team members. In many cases the design team will intentionally deviate from rules of normal form in the data dictionary. This is done when refining the dictionary from the perspective of performance, information preservation in the context of deletes and archives, data availability for user interfaces, user response time, and automation techniques.
In some instances the design materials may include the dialog definitions for specific dialogs where the physical organization of the dialog components is of particular importance to the success of the project. However, in most cases these dialogs are to be constructed by the developers under direction from the "Systems Engineering Guidelines".

As the number of systems we build increasingly expand out to general use on the web, these systems become views of the host companies, presented to groups of users. These systems include artistic content and are characterized by styles, colors, visual or graphic content, sound, and motion. They permit deviations from norms in system-human interactions which the client may see as desirable in order to achieve brand recognition, market distinction, or some other desired property. These are challenging to developers as these elements introduce tremendous source of variability in the ultimate properties of a system. In such projects it is desirable to obtain the client preferences in this area as early as possible, or at least before operational code is integrated with graphic or presentation content. Changing the artistic or graphic content in areas where there is a substantial amount of operational code can substantially increase time requirements and costs. A very good way to address this issue in the design phase is to present to the stakeholders a broad set of graphic options and have them provide a consensus position on what is liked and what is not, what their preferences are across the set of stakeholders. If there is a conflict resolution required, be sure that it is resolved in the favor of the stakeholder(s) who are authorized to provide the final say in graphic content. It is often a very good idea to get this person identified via email or printed correspondence at the beginning of the project.

All data modeling in the design phase is to be performed with ER/Studio. All of the meta data properties are to be fully defined in the logical modeling phase. All model components are fully documented in the design phase. All fields are assigned types, defaults, ranges, accessibility, and are fully described. Associations are fully defined in logical model terms. Entities which have an exclusively “choice list” role are named with the prefix “CL_” to group them and denote their limited purpose. The logical model is to be synchronized for round-trip engineering with the Class Diagrams (and soon the State and Collaboration Diagrams) of GDPro (Describe). The physical model is to be automatically generated from the logical model. The model is to be evaluated with ER/Studio Model Analyzer to evaluate the “correctness” of the model. Data models are modified to meet certain Analyzer objections.

“Describe” (formerly “GDPro”) is to be used to perform all other software systems modeling, providing those UML artifacts needed for the design of the subject system. The Class Diagrams are extended to define the necessary methods for each class. The Use Cases, Activity Diagrams, Statechart Diagrams, Collaboration Diagrams, and Sequence Diagrams are to be constructed and maintained in Describe. Additionally, where needed the Component Diagrams and Deployment Diagrams will be maintained with Describe. The application base Classes for the target development language are to be generated with Describe. These processes will guarantee consistency between the Data Model, the UML Model, and the actual Code.

Prototyping. Many older systems included the development of a prototype in the course of design. These often served as the “working basis” for evolving the target working system. Prototypes are like any other design tool – when used judiciously, where they provide value, they are a great benefit to the development effort. Many times they are used to support an informal software development paradigm of “known when seen”, or “I’ll know it when I see it”. That is, the stakeholder(s) cannot articulate their requirements, but they will know the system which meets them when they see it. This is a very hit or miss approach. It can consume incredible resources (time and money), and has no assurances that the resulting system will actually meet any requirements. Approaches built on this paradigm generally are abandoned. They can be used to articulate the stakeholders’ requirements – although at a very high cost. The greatest risk of using a prototype is that the requirements definition takes a backseat priority to prototype refinement. Stakeholder Requirements are the core of the software development effort, not refinement of work in progress. This type of role of a prototype can easily (and cheaply) be attained by building some of the more important user interfaces (dialogs critical to the workflow of the application) and performing a “walkthrough” with the stakeholders. This provides a means of seeing the interface and making adjustments without the expense of providing all of the dataflow underneath.

The most useful role of prototypes is in risk assessment/abatement. Most useful systems today have risks. Some of these risks are project critical – one or more of the critical objectives of the project cannot be achieved if the risk(s) cannot be overcome. A quick to deploy prototype, constructed and tested early in the project, will provide a great value to the success (or cost aversion) of the project.

Development.

All of the materials which are produced to actualize the target system are an essential part of the development. Most obviously these are the source code, library files, executable file(s) and all of the dependent files. The following is a more complete list of the development materials and what are expected to be included:

1) Installation CD Image – a zip file comprising all components of the installation CD. When unzipped in a target folder, the contents of that folder (and all recursive sub-folders) provides all of the material required to properly install and configure the system on a target workstation, server, or both.
2) All Project or Make files and all files on which these depend.
3) Clear documentation on compilation and link (or build) options which are required for correct reconstruction of the installation components.
4) Clear definition of the tools employed in creating the development components and those required to properly edit the development materials. This should include the manufacturer, product, release, service pack or update, additions to the control set, each library (device) which was used.
5) Clear definition of any support file or reference file or database access required by each subsystem component.
6) Any documents or files which record design refinement, risk resolution, requirement modification, or other decisions or inputs which were made in the development efforts which have not become part of the requirements or design materials.
7) Error/warning/message file.
8) Start up splash screen image file.
9) The user help text file replaced with expanded help text file as the project cycles through text validation and Quality Assurance. This file is over-written during documentation updates cycles, but binding a specific file in early allows better independence and progress of the team.
10) Deployment instructions – these are the explicit details of the deployment instructions. These include the configuration requirements of the target platforms(s) (for example, Microsoft NT Server 4.0 installed as a general server, SP6, IIS 4.0 (Option Pack 2, SP3); MS SQL 7.0, SP2). Any services, which must be running in addition to core services of the identified components, must be listed. If the delivered system is known not to work in specific configurations, these should be listed (e.g. host server must have fixed and published IP address; connection not confirmed through NAT or Proxy Server, or DHCP). The specific release of drivers for target deployment should be listed, particularly if there are known or possible driver instance sensitivities in the application.

Test.

The focus of this section is to verify that the product system meets the specified requirements and validate the system against the design. The primary guidance here must be the requirements. The requirements must be met by the system. How these requirements are met is defined by the Use Cases and the Scenarios defined for each Use Case. This is the first focus of the test cycle. These must be a natural, expeditious, easy-to-perform procedure to execute with the software for each Use Case. Use Cases are often supported directly by a menu option or event. Each Use Case scenario is represented by a possible path (generally through a series of actions or options, buttons or other event centric options) of the event. The software must be tested to verify that each scenario is supported and performed correctly (prescribed actions, state changes, dialogs…) are instantiated in the system. This is the functional test. There is no need to advance to other test issues until the target functionality is achieved as the system will be modified considerably until the functional target is achieved. As soon as testing is commenced a Control Document is created for the system. From that point forward, the same Control Document should be used until testing is "complete". For long projects (more than 6 months) it may be desirable to break out a new Control Document – rarely should there be more than 2. As solved issues are tested and verified they may be merged into the original Control Document. A history of Control issues is extremely important for a number of reasons :

1) Older issues may be re-visited and the decisions made months ago may be completely valid at the current time, or may need to be revised. Being ignorant of the previous design decisions results in thrashing or churning of issues without efficient closure on these issues. It is extremely useful to have the history of issues available in very few places.
2) This provides a single reporting for every issue which was identified in the course of testing. It also provides a small number of places where this documentation might be found.

After the operation has been verified functionally correct and complete, the operation should be tested for technical correctness. This includes the correct computation of dependent values at every step of the operation – the business rules should correctly affect the displayed values and the values which persist in the database. These tests must verify that both sets of values are correct in the context of input values. These should be tested in combination sets and represent the range of required supported values. Included in technical correctness are the proper dispositions of resources that may not be apparent or observable by simply executing the program through its various dynamic paths. These resources include file handles, file or record locks, memory (heap), daemons, threads, and sessions. These resources must be acquired and freed according to the application requirements. While it is impossible to prescribe their use in each case, there are some very useful rules of thumb. File handles should be acquired and freed in the same program unit or in related units called from the same unit (to assure symmetry). Acquiring file handles, assessing file existence and status compatible for the requested rights has performance overhead and should therefore be performed once and then release the handle (close the file) when the operation involving file access is completed. Memory allocation errors and garbage generation has largely been eliminated in C++ by combining the object construction with memory allocation and object destruction with the release of associated memory. Bad style can permit such errors and they should be tested for their absence. File or record locks have guidelines (see "General Systems Engineering Guidelines"). It must be confirmed that the locks are properly freed. Threads and sessions are generally more complex depending on the application and environment. Usually some time interval is identified after which any open thread or session is killed (gracefully).

Once a functional unit (representing a use case) is verified to be functionally complete, it should be tested for soundness, or the "breakability" of the unit should be tested. Most systems constructed to support a set of tasks can often be used in unanticipated ways, often resulting in very undesirable results – heavy systems, memory errors, corrupted database, deadlock,… The system must therefore be tested for sensitivity regarding these issues. Breaking tests include:

1) entering partial data,
2) entering null data,
3) entering illegal data (alpha in numeric fields real values in integer cells, entering more characters than are allowed by field definition, entering special punctuation characters, entering control or escape characters),
4) performing illegal event sequences which are made possible by the system implementation,
5) using enter key instead of tab key,
6) attempting to print without any printers installed,
7) moving control to another program.

It is critical that the system not fail as a result of spurious or erroneous input. Failure in this context would be any of :

1) abnormal termination of program,
2) system hang,
3) database corruption or illegal or erroneous values acquired by one or more fields of a record,
4) incorrect add or delete of a record, inconsistent relational set in the database.

After the above testing is complete there is one last test – the deployment or installation test. The installation disk or CD needs to be taken to an “OS only” machine where no drivers or components from other application installations could be obtained, and walk through the installation instructions by a “new user”. This test also needs to be performed on an “unpopulated” database or “initialized database”. Someone on the team will be able to remember the time when everything was fully tested - “backward and forward”, “every option, with every possible value”, and when the Install CD was sent to the first customers, help desk is asking the developers how does the user log on to the app with an empty User table!! This test cycle is performed to test the Install CD for all of these kinds of issues.

<u>Document.</u>

The user documentation is written to follow and represent the operational properties of the system after the functional requirements and design definition have been validated. All user documentation is for the benefit of the use and utility of the system. It should have a distinct “how-to” flavor, elevating the software to a tool which empowers users in achieving those objectives for which it was intended. The help should be in terms which is familiar and natural to the user community. The documentation should direct the user in a fashion which represents the best or optimal workflow in the context of the users’ workplace and within the natural flow of the system operations. There will always be a distinct and strong dependency between software release and the documentation. If not, then the documentation is not very specific and may, therefore, not be very useful. User documentation should not be vague and it should not simply echo the limited information presented on a dialog in the form of a caption or the text on the status bar. The documentation should be rich with the interdependencies of the system, so that dialog or

system elements are illuminated as members of a grand schema, and not just isolated elements. The documentation should indicate how these interdependencies affect the behavior of system operations or how they interact with other specified elements to achieve certain goals or perform certain services.
The user documentation is produced in an environment of scenario validation – actually using the software while simulating the workplace of the client or target market. It should make the operation within the system a part of the workplace and processes executed in the workplace. There will be operational traits of the software which were not in the design materials or the requirements which will need to be identified and "handled" in the user documentation. There is more wisdom than some understand in the statement "there are no bugs, just undocumented features". There is no such thing as software that is an exact match to a design - no larger or smaller than the design. There are so many properties which are not defined in the design. The software is an actualization of the design and therefore, by definition, an extension of the design. When those features, desirable or not, are documented they become well-defined properties of the system. The goal of documentation is to cast these features in as useful a light as possible. Some side-affects become extensions to the target functionality, through a good application of user documentation.

Systems which have components should have them documented as well. The target of this documentation should be to define the set of components and their configuration dependencies. This might be nothing more than identifying them and stating where they are expected to be found in the file system and registry (where applicable). It may include what services or components must be installed and running on workstations and servers (e.g., Server : NT 4.0 SP5 or later, with MS SQL 7.0 SP 2; Workstation : Win95/98/NT(SP3) with ODBC driver for SQL (3.60 or later), ODBC DB is "Whatchama", user="katskill", password="w7ay9way"). It may also state interdependencies which to the insider may seem insanely obvious, but in 2 months time means the difference between getting a system operational in minutes or having it down for days. It is always easiest to document a system at that point in time where it appears to be the least necessary. When team members are in that special state they should be documenting everything, particularly those things that are so obvious as to hardly needing documentation – the materials will tend to be the most accurate at that time.

Support.

Every system constructed today must be supported. The resources and effort required to support the system are a fundamental design element. A system which is being designed for use by a large population of unsophisticated and inexperienced users, with wide geographic distribution, and no on-site technical support will be very different from a system designed to be used by a relatively small group of highly experienced users with local technical support. Some systems will require onsite training and ongoing help desk support. Some systems will deploy successfully with online help only. Some systems (e.g., embedded systems) will deploy, and run for years, with only one page of a "readme" file. The target support will establish the resources required to provide it. Systems which are built for large populations of inexperienced users which must avoid reliance on support resources must be constructed differently than the same systems with support resources.

Enhance.

Most systems today are “live” systems – they are updated and enhanced over time. Everything about software systems is time dependent. The requirements are defined in a particular time domain. The business model of the organization is that of a particular time. The technologies available for system construction and deployment are time dependent. Very few elements of the system construction are time invariant. Additionally, the arena of software changes quickly and substantially over time. Therefore, it is very reasonable to expect systems to be “live”. By “live” we do not mean “crawling with bugs”. It is the evolution of features which constitutes living systems. The pressure for this evolution is generally the changing business needs of the organizations using the system. Organizations make investments in Systems because they have a superior ROI (Return on Investment). Systems are constructed with priorities defined along the lines of decreasing ROI and complete workflows. These same systems are enhanced under the same dynamics – additional features or feature refinement will provide attractive ROIs, and are therefore worthy of doing.

Summary.

Our task, as full scale developers, is to escort our clients through this maze – the Software Development Process. We are to assist them in everyway we can to define their needs (goals/requirements/constraints), meet or exceed these, and providing visibility of the process along the way in order to maintain their comfort and confidence that their needs are being met. This is achieved through effective communications and documentation throughout the Project. Every Project has peaks and lulls of the communication activity. Typically the first phases are very heavy in communications – Background and Requirements. The remaining phases have heavy communications activities toward the end – Design : Present/Feedback/Refine. A similar profile occurs with Development and Test. We focus on the client requirements and process of systematically meeting those through system feature. If we perform these measures correctly our result is :

- very high quality systems,
- which meet the needs of our clients,
- constructed without waste of time, funding, or talent,
- resulting in very satisfied clients,
- with a tremendous feeling of a “job well done” among all team members.