Real-Time Agility:
Core Principles and Practices for embedded software teams.

By Dr. Bruce Powel Douglass, PhD
Chief Evangelist
IBM Rational

Agile methods are a cohesive set of concepts, principles and practices to address what most consider the banes of software development: poor and changing requirements, short development cycles, long working hours, and burgeoning system complexity. When integrated with the best ideas in design and development using open standards, the result is a full agile development process that can reduce costs and improve quality through the focused application of the key principles. These principles have been applied successfully in many different real-time and embedded markets such as telecommunications, medical, industrial automation, defense and aerospace. This paper introduces practical core principles and practices and shows how they improve project results for embedded system development.

Practice core principles

The IBM Practice Library contains a set of practices that are optimized for the development of software-intensive real time and embedded systems founded on the Harmony for Real-time and Embedded Software Development process. It realizes a set of core principles that apply to an agile approach to embedded development. Understanding these principles will help you understand how the workflows of the practices are organized and why.

Agile core principles

- Your primary goal: develop working software
- You should always measure progress against the goal not against the implementation
- That is, your primary measure of progress is *working software*
- The best way not to have defects in your software is not to put them there in the first place
- Continuous feedback is crucial
- Five Key views of architecture define your architecture
- Plan, Track, and Adapt
- Leading cause of project failure is ignoring risk

The principles of agile, as approached by the Rational Practice Library, focus primarily on quality improvement and risk reduction and only secondarily on team and developer productivity. This is borne out by my experience in the field.

Improving quality also improves productivity through a combination of reduction of rework, and decrease in required retesting.

Core practices

The core principles are enacted by a set of practices – detailed workflows – that guide the work in effective ways.

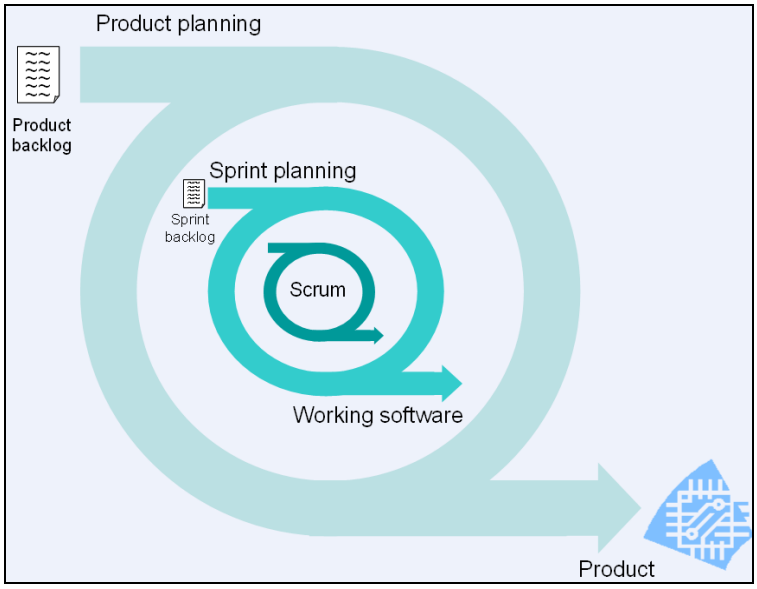

Figure 1: The Agile Workflow

Each practice contains a set of worker tasks, produced or consumed work products, and worker roles and responsibilities. Each practice focuses on a subject matter important for quality management and improvement. In my experience, project failure is primarily the consequence of improperly managed project risk.

Risk is, ultimately, the product of two independent concerns. The first is *severity*, or the impact of a concern on the successful completion of a project. For example, the severity of improperly estimating a schedule can be extremely high, as can reliance on technology that proves ineffective in the target system (such as might occur if it is too slow or resource-intensive). The second is the likelihood of the risk manifesting into a failed project. If we have a history of producing high quality software and the team and technology remain relatively unchanged, the likelihood that defects will be significantly higher is lower than if we have a history of high defect density or if we are radically changing the development team personnel or the product technology.

Risk is all about *stuff you don't actually know* but must to succeed. One of the key aspects of effective risk reduction is that risk should be mitigated as early as possible. This maximizes forward progress and minimizes rework. Pounding out millions of lines of code quickly is not beneficial if that code isn't meeting the customer need or if it is full of defects. As we develop code, we need metrics that assess its quality during development, not in a later testing phase. It is far better to use these practices to avoid putting defects into the code baseline than to spend later effort to track down and repair hundreds or thousands of defects. Most of the practices reduce risk through early or continuous demonstration that the work products being developed are of high quality and meet the need. Put another way, work products that cannot be demonstrated to be of high quality are of little value.

The practices primarily focus on risk reduction although some focus on process improvement. Table 1 shows the most important of the practices, the risks they address, and their mechanism of action.

Table 1: Agile practices for embedded software development

Practice	Risks addressed	Mechanism of Risk Mitigation
Executable Requirements Modeling	• Ambiguous Requirements • Missing Requirements • Conflicting Requirements	• Organize requirements into use cases • Build state-based use case models that demonstrate the interaction of requirements via execution
Model-Based Handoff from Systems Engineering	• Loss of fidelity of information during hand off to software team	• Use principled system engineering model organization to facilitate hand off • Hand Off process migrates logical system engineering work products to physical specifications without loss of fidelity
Dynamic Scheduling	• Schedules are inaccurate • Schedules don't reflect changing project needs or conditions • Schedules don't increase in accuracy during the project	• Use BERT scheduling[1] to get initial best schedule • Create three schedules (Best, Customer, and Goal) • Update schedules frequently founded on outcome-based metrics • Use ERNIE estimation[2] to improve estimation quality • Manage schedules at both project and microcycle levels
High-Fidelity Modeling	• Design doesn't meet requirements need • Design is low quality	• Use UML to model software precisely • Execute and debug models at the model level

[1] ***Real-Time UML: Advances in the UML for Real-Time Systems***, Dr. Bruce Powel Douglass

[2] ***Real-Time UML 3rd Edition: Advances in the UML for Real-Time Systems***, Dr. Bruce Powel Douglass

	• Design is not understood by relevant stakeholders • Code does not accurately reflect design • Architectural views are not reflected in the code	• Use Model-Code Associatively to automatically synchronize models and code • Generate code from models when possible
Continuous Execution & Debugging	• Model and code contain unknown defects	• Execute and debug models at the model level • Use 20-minute nanocycles for model-code-unit test cycles
Test-Driven Development	• Model and code contain unknown defects	• Use 20-minute nanocycles for model-code-unit test cycles • Use model-based testing tooling to create test architectures and execute and analyze test cases
Continuous Integration	• Developers components don't work together properly	• Use daily integrations to ensure different developers implementation works together • Use shared interface libraries (models) for cross-component interfaces and data types
Incremental Development	• System has too many defects • System doesn't meet customer need	• Build the system in 4-6 week microcycles • Perform V&V at the end of each microcycle • Fix any critical defects before moving to next microcycles • Non-critical defects become work items for subsequent microcycles
Continuous Risk Mitigation	• Project fails because of unforeseen problems • Potential catastrophic events and conditions are not tracked	• Create Risk Management Plan with appropriate risk data (e.g. likelihood, severity, owner, mitigation activity) • Schedule risk mitigation activities • Review risk plan and outcomes weekly and at the end of the microcycle
Incremental Process Improvement	• Development team fails to mature and improve their productivity and quality • Key roadblocks to project success are not identified and remediated	• Hold Increment Review at the end of each microcycle and evaluate project (e.g. architecture scalability, schedule adherence, quality metrics, productivity metrics)
Event-Driven Change Management	• Needed changes are forgotten or inadequately addressed • Changes are accepted even though they may be high cost and low value	• Put into place change management system • Evaluate change impact before accepting changes • Update relevant documentation (e.g. requirements, architecture, test plans, schedule) • Assign and track change until

		resolution
On-going Dependability Analysis (safety/ reliability/ security critical projects only)	• System design or implementation is unsafe, unreliable, or insecure	• Perform initial model-based dependability analysis (e.g. UML Safety Analysis Profile, UML Security Analysis Profile) • Revisit analysis during architecture and design to ensure dependability goals are met
Apply Design Patterns for Optimization	• System design isn't coherent • System design isn't understandable • System design is inefficient against relevant design criteria	• Identify optimization criteria • Rank criteria in order of criticality • Identify design solutions (patterns) that optimize the system at acceptable cost • Instantiate design patterns • Test for continued correctness and for optimization goals

Five Key Views of Architecture	• Architecture doesn't meet the system objectives
	• Important architectural aspects are ignored
		• Integrate different views of architecture into the microcycle mission statements
		  • Subsystem architecture
		  • Concurrency architecture
		  • Dependability architecture
		  • Distribution architecture
		  • Deployment architecture
		• Apply design pattern practice at the architectural level

When taking an agile approach to embedded software development, there are three timescales of interest. The overall project is called the *macrocycle* and spans the entire length of the projects. It is constructed of some start up work (generally referred to as "Pre-spiral Planning") and a set of *microcycles.* Each microcycle is 4-6 weeks in length and produces a version of the system on which verification and validation is performed.
Figure 2: Harmony Microcycle

Figure 2 shows the primary activities of the microcycle.

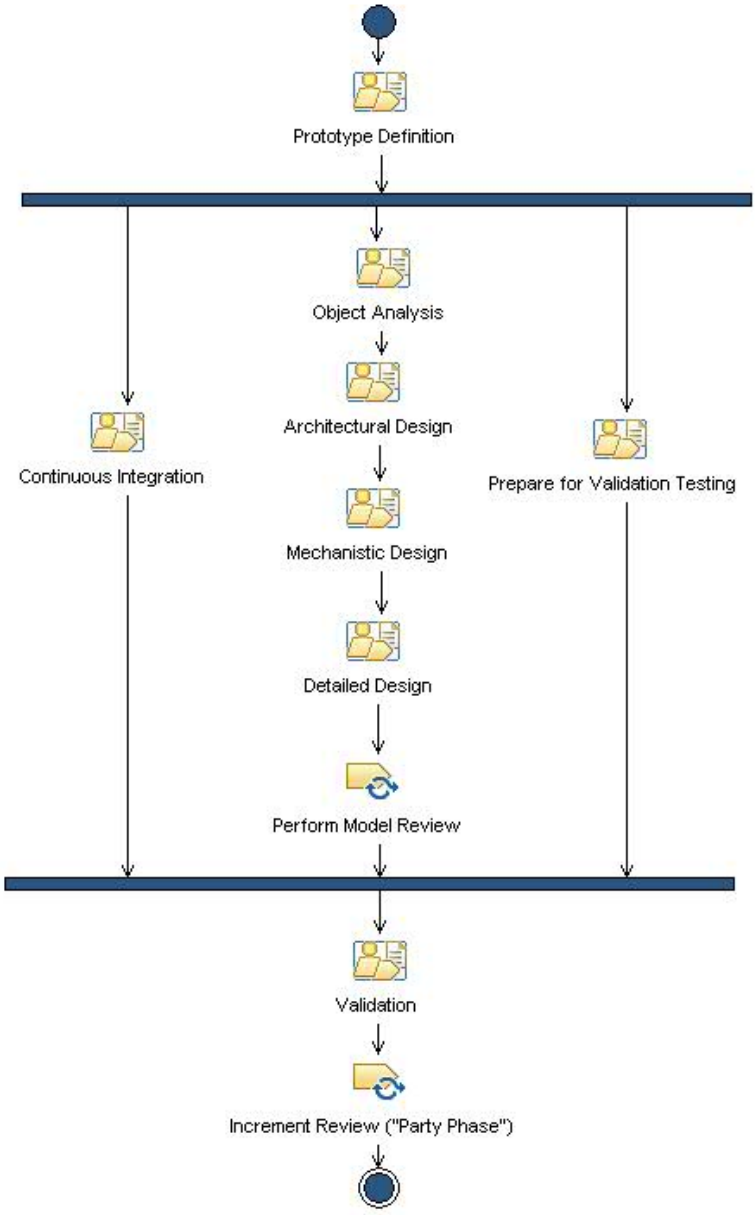

Figure 2: Harmony Microcycle

The smallest unit of time is the *nanocycle*. It is generally between 20 and 60 minutes in length and produces an incremental version of one or more work products that can be assessed for quality. In the Object Analysis nanocycle shown in Figure 3. the inner loops result in executable code and unit tests that are created and applied on a highly frequent basis. This work results in models and code that are not only completely in sync, but are also of very high quality and contain very few defects.

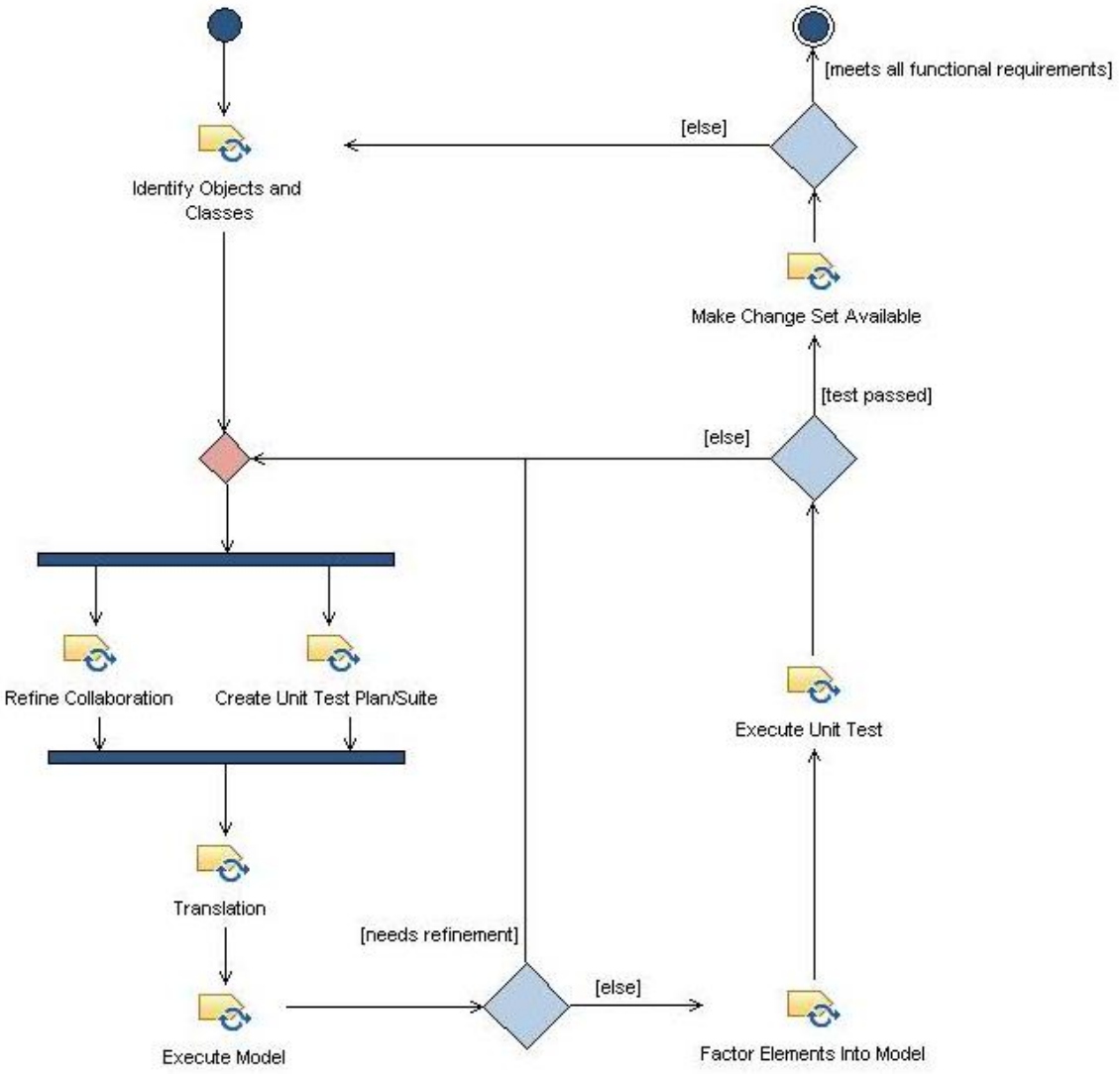

Figure 3: Harmony Nanocycle

Agile Environment with the Rational solution for systems and software engineering

An efficient agile environment provides assistance in doing engineering work using agile practice, facilitates creation and manipulations of work products, aids in worker collaboration, and supports transparent project governance. This requires a tooling environment with work templates generated from process and practice definition, and strong integration of the tools used by the various roles in the systems engineering environment.

The Rational solution for systems and software engineering is integrated on the Rational Jazz platform for agile collaborative system and software delivery. Uniquely attuned to global and distributed teams, Jazz transforms system

delivery by making it more collaborative, productive, and transparent; key aspects of any agile software development environment. You can think of Jazz as an extensible framework that dynamically integrates and synchronizes people, processes, and assets associated with software projects. Unlike the monolithic, closed or point-to-point solutions of the past, Jazz is an open platform that supports the Open Services for Lifecycle Collaboration (OSLC) initiative for improving tool interoperability. Products built on the Jazz platform can leverage a rich set of capabilities for team-based software and systems delivery.

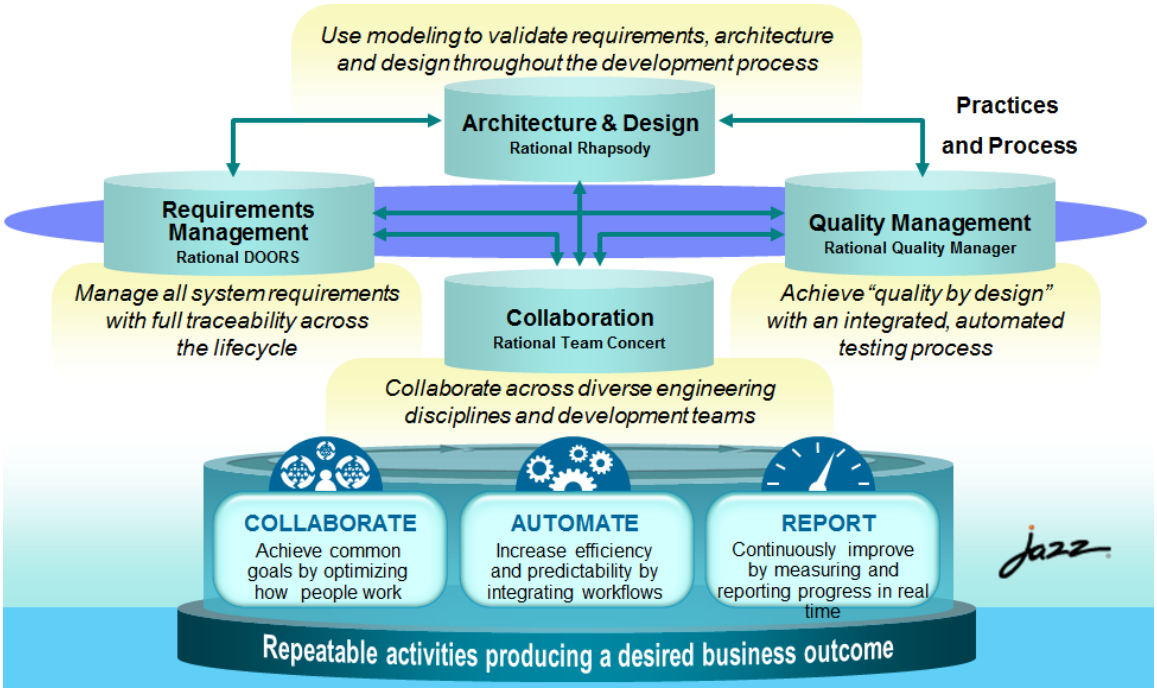

Figure 4: Rational Systems and Software Engineering Solution

Process integration takes place in the Solution in a couple of different ways. First, the Rational Method Composer tool authors process and guidance content, compliant with the Software Engineering Process Metamodel (SPEM) standard. This guidance includes workflows, practices, processes, tasks, work product, worker roles, and various additions such as checklists, concept definitions, examples, and tool mentors. This content is published to a user-consumable form, of which the most common is a web site hosted within the development organization's network.

The second form of process integration is the production of work task templates from the process definition content, that can be used to support planning (e.g. Assign an instance of the "Analyze Use Case" task to Susan) and governance (e.g. reporting on the progress of those assigned tasks). This provides managers and developers with a work environment in which tools, work products, processes plans, and project governance are integrated together seamlessly. This supports project efficiency and visibility across the system delivery lifecycle.

Without integrations across the system delivery lifecycle, system teams are left to operate in silos. When silos form, product delivery effectiveness suffers. In order to deliver smarter products that respond to changing market needs it's required to allow System and Software Engineering teams to perform efficiently and to manage all the lifecycle work products through collaboration. The Rational solution for Systems and Software Engineering provides this integrated system lifecycle management solution.

The Rational solution for systems and software engineering provides an integrated solution for system and software engineering team roles

- *System Engineers*
  The solution provides an integrated and collaborative environment for requirements analysis, architecture management, and work, change and configuration management for teams of systems engineers. The leading products are DOORS and Rhapsody for system engineering tasks, integrated with Team Concert for lifecycle management of the work products. The integrations with Rational Quality Manager provides for strong collaboration with System Validation teams from the start of the project. Work task templates are created from agile system engineering practices to support the assignment of tasks, their execution, and their governance.
- *Project, development and test team leads*
  Rational Team Concert and Quality Manager provides work and plan management for system delivery teams across the project lifecycle and enables live transparency through collaboration, automation, and reporting to the system delivery work products and project health.
- *Software Engineers*
  The Rational solution for systems and software engineering, with Rational Rhapsody integrated with Rational Team Concert in the Eclipse IDE, provides a software development solution for Software Engineers. This integrates model driven development using SysML and UML with the Rational Team Concert capabilities for team collaboration, like model configuration management, work items, change sets, and continuous software build support. The solution also provides traceability to up-stream System Engineering work products in Rational DOORS and Rational Rhapsody, or down-stream traceability to System Integration and Validation.
- *Software and System Testers*
  Rational Quality Manager provides a collaborative environment for test planning, construction, and execution supporting continuous testing as part of the software engineering teams, as well as test management of system validation and acceptance testing. Rational Test Lab Manager improves the efficiency of system test labs and manages how test resources are requested and provided.

The interoperability of the Rational solution for systems and software engineering with other tooling means organizations can add agile capabilities whilst leveraging existing investments in key development infrastructure such as the Rational ClearCase and ClearQuest or the Rational Synergy and Change configuration and change management environments.

Conclusions

Agile methods have their roots in software development. They are a set of practices and technologies meant first to improve the quality of software work products and secondly to improve the productivity of the engineering teams. Agile methods are based on a set of principles that emphasize work that correlates to quality and de-emphasizes activities that do not. The most important of these practices are incremental development, test-driven development, continuous integration, and continuous execution.

Although agile methods come from the IT and enterprise software world, these practices and technologies apply equally well to real-time, embedded software development. Through the use of high-fidelity modeling approaches and organizing the workflow to use the agile practices, the quality of software engineering work products can be significantly improved while lowering the engineering cost.

The recommended approach is to adopt agile methods in an agile way. That is, assess where your organization is and where it wants to be, adopt the technology in an incremental fashion using a pilot project with mentoring from experts, monitor success using KPIs, reassess, and then deploy across your organization.

The Rational solution for systems and software engineering provides a combined tooling and agile practice environment for the various workers within the software intensive systems engineering environment. Strong integration among the key tools in the systems engineering environment including requirements management, architecture and software modeling, quality management and project governance support agile teams in their effort to deliver quality software and systems.

About the author

Bruce Powel Douglass, who has a doctorate in neurocybernetics, has over 30 years experience designing safety-critical real-time applications in a variety of hard real-time environments. He has designed and taught courses in agile methods, object-orientation, MDA, real-time systems, and safety-critical systems development, and is the author of over 5000 book pages from a number of technical books including *Real-Time UML*, *Real-Time UML Workshop for Embedded Systems*, *Real-Time Design Patterns*, *Doing Hard Time*, *Real-Time Agility*, *and Design Patterns for Embedded Systems in C*. He is the Chief

Evangelist at IBM Rational, where he is a thought leader in the systems space, consulting with and mentoring IIBM customers all over the world, represents IBM at many different conferences, and authors tools and processes for the embedded real-time industry. Bruce contributed to both the UML and SysML specifications as well as a number of other standards. He can be followed on Twitter @BruceDouglass. Papers and presentations are available at his Real-Time UML Yahoo technical group (http://tech.groups.yahoo.com/group/RT-UML) and from his IBM page (www-01.ibm.com/software/rational/leadership/thought/brucedouglass.html).

References

Britcher, Robert *The Limits of Software: People, Projects, and Perspectives* (Addison-Wesley, 1999)

Douglass, Bruce Powel *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* (Addison-Wesley, 2002)

Douglass, Bruce Powel *Real-Time Agility: The Harmony/ESW Method for Real-Time and Embedded Systems Development* (Addison-Wesley, 2009)

Douglass, Bruce Powel and Mats, Gothe *IBM Rational Accelerator for Systems and Software Engineering.* http://www.redbooks.ibm.com/abstracts/redp4681.html?Open

Ericson, Clifton II *Hazard Analysis Techniques for System Safety* (Wiley Interscience, 2005)

Hammer, Robert *Patterns for Fault Tolerant Software* (John Wiley and Sons, 2007)

IBM *Smarter Products* *http://www.ibm.com/smarterplanet/us/en/embedded_systems/ideas/index.html?re=sph*

Schumacher, Markus; Fernandez-Buglioni, Eduardo; Hybertson, Duane; Bushmann, Frank; Sommerlad, Peter *Security Patterns: Integrating Security and Systems Engineering* (Wiley and Sons, 2006)