# Social and Technical Reasons for Software Project Failures©

Capers Jones
*Software Productivity Research, LLC*

*Major software projects have been troubling business activities for more than 50 years. Of any known business activity, software projects have the highest probability of being cancelled or delayed. Once delivered, these projects display excessive error quantities and low levels of reliability. Both technical and social issues are associated with software project failures. Among the social issues that contribute to project failures are the rejections of accurate estimates and the forcing of projects to adhere to schedules that are essentially impossible. Among the technical issues that contribute to project failures are the lack of modern estimating approaches and the failure to plan for requirements growth during development. However, it is not a law of nature that software projects will run late, be cancelled, or be unreliable after deployment. A careful program of risk analysis and risk abatement can lower the probability of a major software disaster.*

Software is an important but troubling technology. Software applications are the driving force of modern business operations, but software is also viewed by many chief executives as one of the major problem areas faced by large corporations [1, 2, 3, 4].

The litany of senior executive complaints against software organizations is lengthy, but can be condensed down to a set of three very critical issues that occur over and over in hundreds of corporations:

1. Software projects are not estimated or planned with acceptable accuracy.
2. Software project status reporting is often wrong and misleading.
3. Software quality and reliability are often unacceptably poor.

When software project managers (PMs) themselves are interviewed, they concur that the three major complaints levied against software projects are real and serious. However, from the point of view of software managers, corporate executives also contribute to software problems [5, 6]. The following are three complaints against top executives:

1. Executives often reject accurate and conservative estimates.
2. Executives apply harmful schedule pressure that damages quality.
3. Executives add major new requirements in mid-development.

Corporate executives and software managers have somewhat divergent views as to why software problems are so prevalent. Both corporate executives and software managers see the same issues, but these issues look quite different to each group. Let us examine the root causes of the five software risk factors:

1. Root causes of inaccurate estimating and schedule planning.
2. Root causes of incorrect and optimistic status reporting.
3. Root causes of unrealistic schedule pressures.
4. Root causes of new and changing requirements during development.
5. Root causes of inadequate quality control.

*"One advantage that function points bring to early estimation is that they are derived directly from the requirements and show the current status of requirements completeness."*

These five risk areas are all so critical that they must be controlled if large projects are likely to have a good chance of a successful outcome.

## Root Causes of Inaccurate Estimating and Schedule Planning

Since both corporate executives and software managers find estimating to be an area of high risk, what are the factors triggering software cost estimating problems? From analysis and discussions of estimating issues with several hundred managers and executives in more than 75 companies between 1995 and 2006, the following were found to be the major root causes of cost estimating problems:

1. Formal estimates are demanded before requirements are fully defined.
2. Historical data is seldom available for calibration of estimates.
3. New requirements are added, but the original estimate cannot be changed.
4. Modern estimating tools are not always utilized on major software projects.
5. Conservative estimates may be overruled and replaced by aggressive estimates.

The first of these estimating issues – *formal estimates are demanded before requirements are fully defined* – is an endemic problem which has troubled the software community for more than 50 years [7, 8]. The problem of early estimation does not have a perfect solution as of 2006, but there are some approaches that can reduce the risks to acceptable levels.

Several commercial software cost estimation tools have early estimation modes which can assist managers in sizing a project prior to full requirements, and then in estimating development staffing needs, resources, schedules, costs, risk factors, and quality [9]. For very early estimates, risk analysis is a key task.

These early estimates have confidence levels that initially will not be very high. As information becomes available and requirements are defined, the estimates will improve in accuracy, and the confidence levels will also improve. But make no mistake, software cost estimates performed prior to the full understanding of requirements

are intrinsically difficult. This is why early estimates should include contingencies for requirements changes and other downstream cost items.

The second estimating issue – *historical data is seldom available for calibration of estimates* – is strongly related to the first issue. Companies that lack historical information on staffs, schedules, resources, costs, and quality levels from similar projects are always at risk when it comes to software cost estimation. A good software measurement program pays handsome dividends over time [10].

For those organizations that lack internal historical data, it is possible to acquire external benchmark information from a number of consulting organizations. However, the volume of external benchmark data varies among industries, as do the supply sources.

One advantage that function points bring to early estimation is that they are derived directly from the requirements and show the current status of requirement completeness [11]. As new features are added, the function point total will go up accordingly. Indeed, even if features are removed or shifted to a subsequent release, the function point metric can handle this situation well [12, 13].

The third estimating issue – *new requirements are added but the original estimate cannot be changed* – is that of new and changing requirements without the option to change the original estimate. It is now known that the rate at which software requirements change runs between 1 percent and 3 percent per calendar month during the design and coding stages. Thus, for a project of 1,000 function points and an average 2 percent per month *creep* during design and coding, new features surfacing during design and coding will add about 12 percent to the final size of the application. This kind of information can and should be used to refine software cost estimates by including contingency costs for anticipated requirements creep [14].

When requirements change, it is possible for some projects in some companies to revise the estimate to match the new set of requirements. This is as it should be. However, many projects are forced to attempt to accommodate new requirements without any added time or additional funds. I have been an expert witness in several lawsuits where software vendors were directed by the clients to keep to contractual schedules and costs even though the clients added many new requirements in mid-development.

The rate of requirements creep will be reduced if technologies such as joint application design (JAD), prototyping, and requirements inspections are utilized. Here too, commercial estimating tools can adjust their estimates in response to the technologies that are planned for the project.

The fourth estimating problem – *modern estimating tools are not always utilized on major software projects* – is the failure to use state-of-the-art software cost estimating methods. It is inappropriate to use rough manual *rules of thumb* for important projects. If the costs are likely to top $500,000 and the schedules take more than 12 calendar months, then formal estimates are much safer.

*"Several commercial software cost estimation tools have early estimation modes that can assist managers in sizing the projects prior to full requirements, and then in estimating development staffing needs, resources, schedules, costs, quality, and risk factors."*

Some of the commercial software cost estimating tools used in 2006 include: COCOMO II, Construx Estimate, COSTAR, CostXpert, KNOWLEDGEPLAN, PRICE-S, SEER, SLIM, and SOFTCOST.

For large software projects in excess of 1,000 function points, any of these commercial software cost estimating tools can usually excel manual estimates in terms of accuracy, completeness, and the ability to deal with tricky situations such as staffing buildups and growth rate in requirements.

Estimating tools have one other major advantage: when new features are added or requirements change, redoing an estimate to accommodate the new data usually only takes a few minutes. In addition, these tools will track the history of changes made during development and, hence, provide a useful audit trail.

The fifth and last of the major estimating issues – *conservative estimates may be overruled and replaced by aggressive estimates* – is the rejection of conservative or accurate cost estimates and development schedules by clients or top executives. The conservative estimates are replaced by more aggressive estimates that are based on business needs rather than on the capabilities of the team to deliver. For some government projects, schedules may be mandated by Congress or by some outside authority. There is no easy solution for such cases.

The best solution for preventing the arbitrary replacement of accurate estimates is evaluating historical data from similar projects. While estimates themselves might be challenged, it is much less likely that historical data will be overruled.

It is interesting that high-tech industries are usually somewhat more sophisticated in the use of estimating and planning tools than financial services organizations, insurance companies, and general manufacturing and service groups. The high-tech industries such as defense contractors, computer manufacturers, and telecommunication manufacturers need accurate cost estimates for their hardware products, so they usually have estimating departments that are fully equipped with estimating tools that also use formal estimating methods [15].

Banks, insurance companies, and *low-technology* service companies do not have a long history of needing accurate cost estimates for hardware products so they have a tendency to estimate using informal methods and also have a shortage of estimating tools available for software PMs.

## Root Causes of Incorrect and Optimistic Status Reporting

One of the most common sources of friction between corporate executives and software managers is the social issue that software project status reports are not accurate or believable. In case after case, monthly status reports are optimistic that all is on

schedule and under control until shortly before the planned delivery when it is suddenly revealed that everything was not under control and another six months may be needed.

What has long been troubling about software project status reporting is the fact that this key activity is severely underreported in software management literature. It is also undersupported in terms of available tools and methods.

The situation of ambiguous and inadequate status reporting was common even in the days of the *waterfall model* of software development. Inaccurate reporting is even more common in the modern era where the *spiral model* and other alternatives such as *agile methods* and the object-oriented paradigm are supplanting traditional methods. The reason is that these nonlinear software development methods do not have the same precision in completing milestones as did the older linear software methodologies.

The root cause of inaccurate status reporting is that PMs are simply not trained to carry out this important activity. Surprisingly, neither universities nor many in-house management training programs deal with status reporting.

If a project is truly under control and on schedule, then the status reporting exercise will not be particularly time consuming. Perhaps it will take five to 20 minutes of work on the part of each component or department manager, and perhaps an hour to consolidate all the reports.

But if a project is drifting out of control, then the status reports will feature *red flag* or warning sections that include the nature of the problem and the plan to bring the project back under control. Here, more time will be needed, but this is time very well spent. The basic rule of software status reporting can be summarized in one phrase: No surprises!

The monthly status reports should consist of both quantitative data on topics such as current size and numbers of defects and also qualitative data on topics such as problems encountered. Seven general kinds of information are reported in monthly status reports:

1. Cost variances (quantitative).
2. Schedule variances (quantitative).
3. Size variances (quantitative).
4. Defect removal variances (quantitative).
5. Defect variances (quantitative).
6. Milestone completions (quantitative and qualitative).
7. Problems encountered (quantitative and qualitative).

Six of these seven reporting elements are largely quantitative, although there may also be explanations for why the variances occur and their significance.

The most common reason for schedule slippage, cost overrun, and outright cancellation of a major system is that they contain too many bugs or defects to operate successfully. Therefore, a vital element of monthly status reporting is recording data on the actual number of bugs found compared to the anticipated number of bugs. Needless to say, this implies the existence of formal defect and quality estimation tools and methods.

Not every software project needs the rigor of formal monthly status reporting. The following kinds of software need monthly status reports:

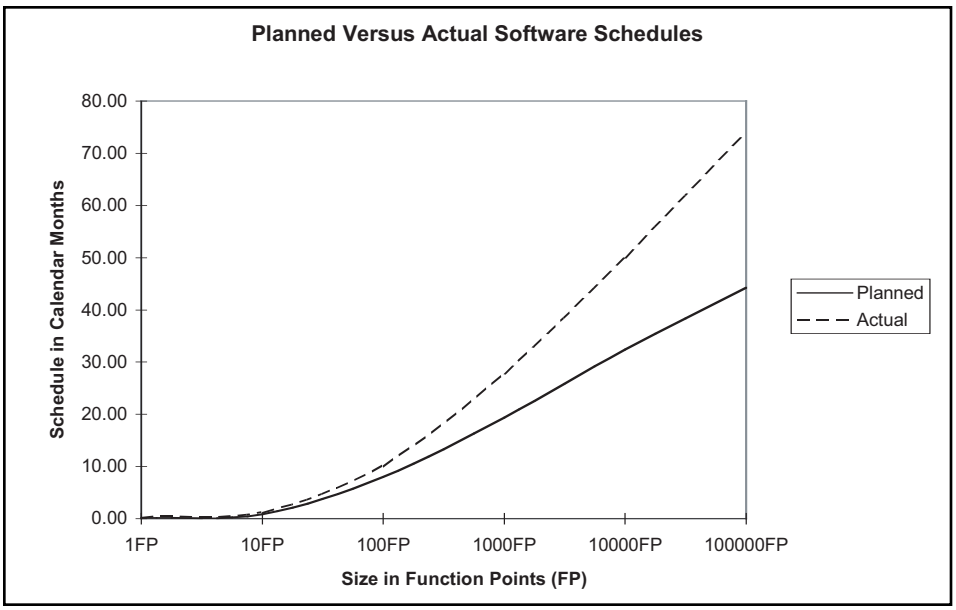
- Projects whose total development costs are significant (>$1,000,000).
- Projects whose total development schedule will exceed 12 calendar months.
- Projects with significant strategic value to the enterprise.
- Projects where the risk of slippage may be hazardous (such as defense projects).
- Projects with significant interest for top corporate management.
- Projects created under contract with penalties for non-performance.
- Projects whose delivery date has been published or is important to the enterprise.

The time and effort devoted to careful status reporting is one of the best software investments a company can make. This should not be a surprise: status reports have long been used for monitoring and controlling the construction of other kinds of complex engineering projects.

During the past 20 years, a number of organizations and development approaches have included improved status reporting as a basic skill for PMs. Some of these include the Project Management Institute, the Software Engineering Institute's (SEI) Capability Maturity Model® (CMM®), the reports associated with the Six Sigma quality methodology, and the kinds of data reported when utilizing International Organization for Standardization (ISO) Standards.

Unfortunately, from examining the status reports of a number of projects that ended up in court for breach of contract, inaccurate status reporting still remains a major contributing factor to cost overruns, schedule overruns, and also to litigation if the project is being performed under contract.

Figure 1: *Planned Versus Actual Schedules for Software Projects*




## Root Causes of Unrealistic Schedule Pressures

Unrealistic schedule pressure by executives or clients is a common software risk factor. There are four root causes for unrealistic schedule pressure:

1. Large software projects usually

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

have long schedules of more than 36 months.
2. PMs are not able to successfully defend conservative estimates.
3. Historical data from similar projects is not available.
4. Some kind of external business deadline affects the schedule.

Figure 1 shows U.S. industry experiences derived from several thousand software projects. The upper curve shows the average delivery time in calendar months, while the lower curve shows the planned or desired delivery time. The larger the project, the greater the gap between the actual delivery date and the planned delivery date of the application [16].

Executives may be arbitrary in their decisions, but they are seldom stupid. While corporate executives might want a large software project finished in 24 months, they will almost certainly accept a 36-month schedule as a fact of life (if they know that not one of 50 similar projects within their industry has ever been completed in less than 36 months). Estimates might be overruled, but accurate historical data will probably keep schedule pressure from becoming unrealistic.

The most difficult problem to solve is when some kind of external business deadline affects the project schedule. Unfortunately, these business deadlines are usually outside the control of either PMs or technical personnel. Examples of external business deadlines include contractual obligations, the starting dates of new laws that require software support, or some kind of technical situation such as those associated with the Y2K problem.

Such external fixed dates cannot be changed, or at least not changed by project personnel. Therefore a combination of cutting back on functions, plus staff overtime, remains the most common method for dealing with fixed and unchanging delivery dates. If the mandated schedule is quite impossible to achieve, then a more drastic option would be project cancellation.

## Root Causes of New and Changing Requirements During Development

The root causes of requirements changes are dynamic businesses. Real-world requirements for software must change in response to new business needs. However, average change rates of 2 percent per calendar month indicate that the methods used for gathering and analyzing the initial requirements are inadequate and should be improved.

By counting function points from the original requirements and then counting again at the time of delivery, it has been found that the average rate of requirements growth is about 2 percent per calendar month from the nominal completion of the requirements phase through the design and coding phases.

The total accumulated volume of new or changing requirements can top 50 percent of the initial requirements when function point totals at the requirements phase are compared to function point totals at deployment. The state-of-the-art requirements change control includes the following:

*"More than 50 years of empirical studies have proven that projects with effective quality control cost less and have shorter schedules than similar projects with poor quality control."*

- A joint client/development change control board.
- Use of JAD to minimize downstream changes.
- Use of formal prototypes to minimize downstream changes.
- Formal review of all change requests.
- Revised cost and schedule estimates for all changes under 50 function points.
- Prioritization of change requests in terms of business impact.
- Formal assignment of change requests to specific releases.
- Use of automated change control tools with cross-reference capabilities.

One of the observed byproducts of the usage of formal JAD sessions is a reduction in downstream requirements changes. Rather than having unplanned requirements surface at a rate of 1 percent to 3 percent every month, studies of JAD by IBM and other companies have indicated that unplanned requirements changes often drop below 1 percent per month due to the effectiveness of the JAD technique.

Prototypes are also helpful in reducing the rates of downstream requirements changes. Normally, key screens, inputs, and outputs are prototyped so users have some *hands-on* experience with an example of the completed application.

However, changes will always occur for large systems. It is not possible to freeze the requirements of any real-world application.Therefore, leading companies are ready and able to deal with changes and do not let them become impediments to progress; some form of iterative development is a logical necessity.

## Root Causes of Inadequate Quality Control

Effective software quality control is the most important single factor that separates successful projects from delays and disasters. The reason for this success is that finding and fixing bugs is the most expensive cost element for large systems, and it takes more time than any other activity.

The root cause for poor quality control is lack of solid empirical data on the cost effectiveness of a good quality control program. More than 50 years of empirical studies have proven that projects with effective quality control cost less and have shorter schedules than similar projects with poor quality control. However, a distressing number of PMs are not aware of the economics of quality control [5, 10].

Successful quality control involves defect prevention, defect removal, and defect measurement activities. The phrase *defect prevention* includes all activities that minimize the probability of creating an error or defect in the first place. Examples of defect prevention activities include the use of the Six Sigma approach, the use of JAD for gathering requirements, the use of formal design methods, the use of structured coding techniques, and the use of libraries of proven reusable material.

The phrase *defect removal* includes all activities that can find errors or defects in any kind of deliverable. Examples of defect removal activities

include requirements inspections, design inspections, document inspections, code inspections, and many kinds of testing [17, 18].

The phrase *defect measurement* includes measures of defects found during development and also defects reported by customers after release. These two key measures allow leading companies to calculate their defect removal efficiency rates, or the percentages of defects found prior to release of software applications. Supplemental measures such as severity levels, code complexity, and defect repair rates are also useful and important. Statistical analysis of defect origins and root-cause analysis are beneficial, along with the key measurements of cost and defect repairs [10].

Some activities benefit both defect prevention and defect removal simultaneously. For example, participation in design and code inspection is very effective in terms of defect removal and also benefits defect prevention. Defect prevention is aided because inspection participants learn to avoid the kinds of errors that inspections detect.

Successful quality control activities include defect prevention, defect removal, and defect measurements. The combination of defect prevention and defect removal activities leads to some very significant differences in the overall numbers of software defects between successful and unsuccessful projects.

For projects in the 10,000 function point range, the successful ones accumulate development totals of around 3.0 defects per function point and remove about 96 percent of them before customer delivery. In other words, the number of delivered defects is about 0.12 defects per function point or 1,200 total latent defects. Of these, about 10 percent – or 120 – would be fairly serious defects. The rest would be minor or cosmetic defects.

By contrast, the unsuccessful projects accumulate development totals of around 7.0 defects per function point and remove only about 85 percent of them before delivery. The number of delivered defects is about 1.05 defects per function point or 10,500 total latent defects. Of these, about 15 percent – or 1,575 – would be fairly serious defects. This large number of serious latent defects after delivery is very troubling for users. If a project has more than about 7.0 defects per function point and less than 85 percent removal efficiency, it will probably be cancelled because it can never successfully exit testing, and the test cycle will be hopelessly protracted.

One of the reasons why successful projects have such a high defect removal efficiency compared to unsuccessful projects is the use of design and code inspections [17, 18]. Formal design and code inspections average about 65 percent efficient in finding defects. They also improve testing efficiency by providing better source material for constructing test cases.

Unsuccessful projects typically omit design and code inspections and depend purely on testing. The omission of upfront inspections causes three serious problems:

1. The large number of defects still present when testing begins slows the project to a standstill.
2. The *bad fixes*[1] injection rate for projects without inspections is alarmingly high.

*"The most common reason for schedule slippages, cost overruns, and outright cancellation of major systems is that they contain too many bugs or defects to operate successfully."*

3. The overall defect removal efficiency associated with only testing is not sufficient to achieve defect removal rates higher than about 80 percent.

Fortunately, the SEI, ISO quality standards, and the Six Sigma approach have benefited quality control activities throughout the past 20 years. As a result, an increasing number of large projects have been successful compared to similar projects done in the 1980s.

However, for very large projects above 10,000 function points in size, missed delivery dates, cost overruns, and outright terminations remain distressingly high even in 2006. The industry is improving, but much more improvement is needed.

## Summary and Conclusions

Large software projects are very hazardous business ventures. For projects above 10,000 function points, cancellations, delays, and cost overruns have been the norm rather than the exception.

Careful analysis of the root causes of large software project delays and disasters indicate that most of the problems stem from inaccurate estimation, inaccurate status reporting, lack of historical data from similar projects, and suboptimal quality control.

All of these root causes can be minimized or even eliminated by the adoption of formal estimating methods and tools, formal monthly status reports of both quantitative and qualitative data, collecting historical data, and improving quality control methods. Large software projects will never be without risk, but if the risks can be brought down to acceptable levels, both clients and corporate executives will be pleased.◆

## Note

1. The term *bad fixes* refers to secondary defects accidentally injected by means of a patch or defect repair that is itself flawed. The industry average is about 7 percent, but for unsuccessful projects the number of bad fixes can approach 20 percent; i.e. one out of every five defect repairs introduced fresh defects [14]. Successful projects, on the other hand, can have bad-fix injection rates of only 2 percent or less.

## References


1. Yourdon, Ed. Death March – The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects. Upper Saddle River, NJ: Prentice Hall, 1997.
2. Glass, R.L. Software Runaways: Lessons Learned from Massive Software Project Failures. Prentice Hall, 1998.
3. Johnson, James. "The Chaos Report." West Yarmouth, MA: The Standish Group, 2000.
4. Ewusi-Mensah, Kweku. Software Development Failures. Cambridge, MA: Massachusetts Institute of Technology Press, 2003.
5. Jones, Capers. Assessment and Control of Software Risks. Prentice Hall PTR, 1994.
6. Jones, Capers. Patterns of Software System Failure and Success. Boston, MA: International Thomson Computer Press, 1995.
7. Boehm, Barry. Software Engineering Economics. Englewood Cliffs, NJ: Prentice Hall, 1981.
8. Jones, Capers. "Sizing Up Software." Scientific American Magazine Dec. 1998: 104-111.
9. Jones, Capers. Estimating Software

Costs. New York, NY: McGraw Hill, 1998.
10. Kan, Stephen H. Metrics and Models in Software Quality Engineering. 2nd ed. Boston, MA: Addison-Wesley Professional, 2002.
11. Jones, Capers. Applied Software Measurement. 2nd ed. New York, NY: McGraw Hill, 1996.
12. Garmus, D. and D. Herron. Function Point Analysis – Measurement Practices for Successful Software Projects. Boston, MA: Addison-Wesley Professional, 2001.
13. International Function Point Users Group (IFPUG). IT Measurement – Practical Advice from the Experts. Boston, MA: Addison-Wesley, 2002.
14. Jones, Capers. Software Quality – Analysis and Guidelines for Success. Boston, MA: International Thomson Computer Press, 1997.
15. Jones, Capers. Software Assessments, Benchmarks, and Best Practices. Boston, MA: Addison-Wesley Professional, 2000.
16. Jones, Capers. Conflict and Litigation Between Software Clients and Developers. Narragansett, R.I.: Software Productivity Research LLC, 2005.
17. Radice, Ronald A. High Quality, Low Cost Software Inspections. Andover, MA: Paradoxicon Publishing, 2002.
18. Wiegers, Karl E. Peer Reviews in Software – A Practical Guide. Boston, MA: Addison Wesley Professional, 2002.

## About the Author

**Capers Jones** is currently the chairman of Capers Jones & Associates, LLC. He is also the founder and former chairman of Software Productivity Research, LLC (SPR), where he holds the title of Chief Scientist Emeritus. He is a well-known author and international public speaker, and has authored the books "Patterns of Software Systems Failure and Success," "Applied Software Measurement," "Software Quality: Analysis and Guidelines for Success," "Software Cost Estimation," and "Software Assessments, Benchmarks, and Best Practices." Jones and his colleagues from SPR have collected historical data from more than 600 corporations and more than 30 government organizations. This historical data is a key resource for judging the effectiveness of software process improvement methods. The total volume of projects studied now exceeds 12,000.

**Software Productivity Research, LLC**
**Phone: (877) 570-5459**
**(973) 273-5829**
**Fax: (781) 273-5176**
**E-mail: cjones@spr.com**

# CALL FOR ARTICLES

If your experience or research has produced information that could be useful to others, CROSSTALK can get the word out. We are specifically looking for articles on software-related topics to supplement upcoming theme issues. Below is the submittal schedule for three areas of emphasis we are looking for:


**Enabling Technologies for Net-Centricity**
*January 2007*
Submission Deadline: August 21

**Agile Development**
*February 2007*
Submission Deadline: September 18

**COTS Integration**
*March 2007*
Submission Deadline: October 16

**Please follow the Author Guidelines for CROSSTALK, available on the Internet at <www.stsc.hill.af.mil/crosstalk>. We accept article submissions on all software-related topics at any time, along with Letters to the Editor and BackTalk.**

## COMING EVENTS

**July 5-7**
*18th International Conference on Software Engineering and Knowledge*
San Francisco, CA
www.ksi.edu/seke/seke06.html

**July 6-8**
*SEDE 2006*
*15th International Conference on Software Engineering and Data Engineering*
Los Angeles, CA
www.isp.mu-luebeck.de/sede06/index.htm

**July 16-19**
*WMSCI 2006*
*The 10th World Multi-Conference on Systemics, Cybernetics, and Informatics*
Orlando, FL
www.iiisci.org/wmsci2006/website/default.asp

**July 16-19**
*SERP 2006*
*The 3rd Symposium on Risk Management and Cyber-Informatics*
Orlando, FL
www.iiisci.org/rmci2006/website/default.asp

**July 23-28**
*Agile 2006*
Minneapolis, MN
www.agile2006.com

**July 24-28**
*Practical Software and Systems Measurement (PSM)*
*10th Annual Users' Group Conference*
Vail, CO
www.psmsc.com/Events.asp

**April 16-19, 2007**
*2007 Systems and Software Technology Conference*
www.sstc-online.org