

# Experience With A Process For Competitive Programming

Bart Massey  
Computer Science Department  
Portland State University  
bart@cs.pdx.edu

*A lightweight software development process was developed for the unusual programming environment of the Association for Computing Machinery International Collegiate Programming Contest. This process was used by teams from Portland State University over the course of two years, with excellent results: the process was judged to be effective in improving the performance and satisfaction of student programmers. The process modeling techniques and process elements used should be portable to a wide range of industrial domains for which standard methodologies are inappropriate, especially to smaller teams and projects.*

Bart Massey received a B.A. in Physics from Reed College, then spent several years as a Software Engineer III in the Television Measurement Systems Division of Tektronix, Inc. Returning to school to study programming languages and artificial intelligence, Bart received his Ph.D. in 1999 from the University of Oregon. He has spent the last few years teaching at Portland State University and in the Oregon Master of Software Engineering program.

## 1 The ACM ICPC

The Association for Computing Machinery (ACM) has conducted its International Collegiate Programming Contest (ICPC) [1] on an annual basis since 1970. In 2000, more than 2,400 teams of students were involved in this competition at the regional level, making the 2000 ICPC arguably the largest organized student programming experience in history.

Internationally competitive student ICPC teams tend to be composed of the best and brightest programmers at the top-ranked schools in the world. This tendency is enhanced by the unusual structure of the competition, little-changed since the 1970s. In brief, a team of three students (at most one having an undergraduate degree) shares a single terminal over a period of around 8 hours. The team is given 6 or more “brief” requirement specifications in a stylized format at the start of competition. They are then expected to cooperatively analyze the problems, select the most tractable (there is rarely sufficient time for even the top teams to solve all the problems) and to produce programs which will produce correct answers given blind test inputs. Scoring is by number of problems solved, with ties broken using a combination of time-to-solution and number of incorrect submissions.

## 2 Teamwork and the ICPC

The ICPC contest format is supposed to encourage team-based problem solving, but it often has the opposite effect. In practice a single expert programmer, though monopolizing the terminal and receiving little or no support from teammates, is often more effective than a coherent team. Like professional athletic teams or small industrial software development teams, a dominant player can make the team effort a one-person show. There are several reasons why the ICPC tends to exaggerate rather than minimize this problem [2].

First, the well-known wide gap in productivity between programmers [5] appears to hold in this situation. Since the contest time period is so short and access to the terminal is a fairly effective bottleneck, this productivity gap is amplified: the effectiveness of the programmer at the terminal tends to limit the progress of the team. Thus, on a competent team, an exceptional programmer will dominate programming activity.

Second, the usual problems of effective programming team communication are exacerbated by the contest conditions. While software engineering communications problems have been appreciated at least since the mid-70s [5], the ICPC’s short time scale and limited access to computing, display, and typographic resources all aggravate the situation. Currently, students have been encouraged by modern programming environments and practices to think online and share their thoughts via e-mail and the web: this leaves them ill-prepared for the high-pressure isolated pencil-and-paper problem solving environment of the ICPC.

Third, the methods of contest preparation and organization commonly used by the student teams and their advisors tend to encourage individualization rather than teamwork. For starters, it is common for an institution to select participating team members based on their placement in an individual competition under contest conditions. While this method undoubtedly helps select those with a certain skill set, it is likely to select against those who work best cooperatively. Further, there appears to be a strong tendency among coaches to emphasize parallel problem solving during the competition: this typically manifests as a suggestion that the problems be divided up among team members, with each team member working on a problem independently [6]. Again, while the benefits of this approach are obvious, the less obvious downside is that a team member’s attempt to elicit help from or offer advice to another team member may be treated as a source of inefficiency and actively discouraged.

### 3 A Process Model For ICPC Preparation and Participation

In the Fall of 1999 the author was charged with preparing teams from Portland State University (PSU) in Portland, Oregon for the Pacific Northwest (PNW) Regional ICPC. Recent PSU experience with the Regional ICPC competition had not been good: the two 1997 teams had solved one problem each, and in 1998 one team solved two problems while the other failed to solve a single problem. This was particularly disappointing given that at least one problem in a typical year is designed to be especially easily soluble. In addition to a demoralizing past, the time available was short: work began about one month before the competition date.

Finally, the quality of the students was fairly uniform: while all of the contestants were quite competent, there was no single “superstar” programmer. (This is a common situation in industrial software development as well: extraordinary performers are scarce and expensive. Approaches that work well in this setting should thus be useful in the industrial environment as well as the academic one.)

Given these constraints, a new approach to contest preparation and execution was needed. The approach selected was predicated on achieving significant improvement quickly using a uniform talent pool. An informal software development process was sketched out, with the goal of allowing (and indeed enforcing) effective teamwork under contest conditions. The major components of this model were:

- A structured process for evaluation and ordering of problems to be attempted. Teams were instructed to
  - Read through each problem as a group.
  - Discuss any ambiguities, incompleteness, or inconsistency in the problem specification.
  - Discuss the general approaches that might be used to solve the problem.
  - Sketch a high-level architecture for a solution program.
  - Estimate (in an *ad hoc* fashion) time and lines-of-code for each architectural block, and then sum these estimates to achieve an estimate for the entire program.
  - Find whether there was a “champion” for the problem on the team; someone excited about attacking it.

Using this process, the problems were to be ordered (by team consensus, with the team captain having veto power) and tackled sequentially. Based on estimates by the coach and experience during practice sessions, the eventual rule of thumb was that this portion of the overall process should take roughly 30-60 minutes.

- A structured process for tackling each problem. The three team members were instructed to apportion the roles of *requirements analyst*, *implementer*, and *tester* among them. The requirements analyst was instructed to carefully analyze the requirements and design, and was also charged with proofreading and assisting with coding. The regional contest rules allowed at most two team members at each terminal simultaneously: the third team member was excluded. This third team member, the tester, was charged with developing black-box and gray-box system test cases from the information available, and delivering these at the appropriate time. Only when all test cases passed, and all team members concurred that the program was complete and correct, was the program to be submitted for judging.

An instructional approach to contest preparation was developed concurrently. There were several components deemed important to emphasize:

**Problem Classification:** ICPC problems tend to fall into a small number of recognizable categories. Some of these categories contain a large percentage of deceptively difficult or time-consuming problems.

Others require specialized knowledge or approaches (that the students may not possess) to solve effectively. Some time was spent training the students to classify problems and re-evaluate their difficulty.

Similar situations arise in anticipating problematic components and techniques in small industrial software development projects. One of the most important contributions of an experienced developer is the ability to recognize and leverage these problem patterns: team training is a key component of this.

**Requirements Specification:** An ICPC “programming problem” description is invariably an English-language requirements document. The quality of construction and proofreading varies widely: a good rule of thumb is that the problem requirements must always be considered carefully. Some effort was made to teach students to look for ambiguous, incomplete, or contradictory requirements, and to explain techniques for dealing with these defects under contest conditions.

The successful deconstruction of requirements documents is a skill that any good software development team must possess. While the ICPC “requirements document” is somewhat idiosyncratic in structure and form, its principal defects are shared with the majority of requirements specifications encountered in industrial situations, and the approaches to detecting and correcting these defects are quite similar.

**Debugging:** Incredible as it may seem, in the author’s experience even highly effective student programmers generally evince little knowledge of effective debugging techniques and practices. Indeed, it seems plausible that the often-deplored rapid “edit-compile-crash” cycle common among student programmers results less from the ease of access to high-speed editing and compilation than from simple lack of education in effective debugging practice. The students were taught the rudiments of a formal debugging methodology, and observed under contest conditions to evaluate their effectiveness in the use of this methodology.

The lack of methodology and expertise in debugging does not magically disappear once a student graduates. Given the crucial importance of this skill to the implementation and testing phases of software development, more attention should be paid in both academia and industry to education about effecting debugging process.

**Testing:** Because the contestants are penalized for incorrect submissions, it is important to submit a program that works correctly the first time. Further, because the requirements are often problematic and no suite of adequate test cases is provided as part of the contest materials, on-line integration and system test development is absolutely crucial to successful detection and diagnosis of program defects. The participants were instructed in the basics of informal test construction and evaluation [7].

Small team software development projects usually lack the resource of an effective independent testing group. In this situation, it is highly important that the implementors construct effective tests, and use a systematic test execution and evaluation process. The benefits of early defect detection and high-quality error information should more than pay for the extra effort.

As important as the emphasis on some phases of the software lifecycle is the de-emphasis of others. Of particular note is the near-total lack of attention to actual implementation. What instruction there was in this regard focused principally on fundamentals of readability and on the unusual topic of how to sacrifice efficiency for correctness and brevity during implementation.

Because of the poor computing infrastructure and limited control of programming language and environment available under contest conditions, unit and integration testing were deemed to be of limited value. Students were encouraged to unit test particularly problematic procedures or functions, but this encouragement was supplemented by little guidance or assistance.

## 4 Experience With The Process Model

In the 1999 PNW Regional ICPC, the two PSU teams each solved 3 of 8 problems, placing in the top half of the 61 schools participating. Students present on both 1998 and 1999 teams reported subjectively that the process employed was crucial to their improved performance. In particular, a marked improvement in estimation and time management was felt to greatly improve teamwork, leading to serious synergy in solution.

Post-mortem analysis suggested some minor improvements to the process, notably better team partitioning techniques and a slightly different allocation of team members to roles. Somewhat surprisingly, the emphasis on up-front activities in the model was endorsed by the students: they felt that accurate problem selection was key to their increased success.

Based on this feedback, the model was revised to include a limited amount of overlapping work on problems. In essence, while two team members (the implementor and the tester) finished up the implementation and submission, the other team member would become the principal implementor on the next problem. This was felt to be a bit more efficient. In addition, more emphasis was paid to careful testing and debugging discipline, which is difficult to enforce under the pressure of an actual contest.

Preparation for the 2000 contest began somewhat earlier in the academic year. One result of this earlier start and of the 1999 success was that the student pool was larger, and not all interested students were able to participate at the regional level. The final team composition was initially set by “coach’s fiat”, but was then altered by negotiations between team members. The resulting teams seemed to contain an effective combination of skills and experience.

The 2000 PNW Regional ICPC problems were substantially harder than those of 1999. For example, in 1999 the winning teams solved 8 of 8 problems, while in 2000 the top team solved 5 of 8. Similarly, in 1999, only 4 of the 61 teams participating solved no problems, while in 2000 10 of the 57 teams failed to achieve a single solution. The experience of the PSU 2000 teams was similar: each team solved 2 of 8 problems. This once again put them in roughly the middle of the overall ranking.

Several things were notable about the 1999 and 2000 PSU teams experiences. First, they solved their first problem substantially later than other teams with comparable records. Indeed, in 2000 only one problem was solved by either PSU team with only two hours remaining in the competition. This statistic seems to accurately reflect the extra up-front work put in, and this delay seems to have payed off in improved overall performance.

Second, the inter-team performance was remarkably consistent. This may be due to chance and the small sample size, but it seems intuitively to be due to the similar careful preparation and similar skill levels of all team members.

## 5 Application To Industrial Practice

Basic software process engineering has produced a software development process which seems to be reasonably effective in a highly unusual development environment. While the small sample size and unscientific nature of the experiment make it impossible to draw definitive statistical conclusions, a few modest observations seem appropriate.

The objective and subjective results seem to reflect truisms of the software process engineering community. First, software process was not a “silver bullet” [4]: the PSU teams did not become indomitable software power-houses. A fairly dramatic improvement was produced, but from the lower range of performance to the middle range. Second, while the improvements produced did not seem to be merely a function of greater resource availability or better-quality personnel, they were conditioned by more careful attention. Third, effective process engineering was not inordinately difficult: substantial gains were achieved by simple

and relatively common-sense methods. Finally, good software process was not necessarily unpalatable to those involved: in this case, student volunteers enthusiastically endorsed the increased focus and discipline associated with organized software development.

Much of the literature on software process and productivity is targeted at large organizations working on a series of large-scale and often safety-critical or mission critical software projects. Where attention has been paid to software process components and customization, it has usually been to techniques designed to scale well and to regularize the use of known best-practice techniques. There are several good reasons for this, notably the more extreme need for good process in development on this scale, and the fact that the whole notion of process-based development is borrowed from large-scale industrial engineering.

However, the gap between this sort of large-scale development process and the perhaps more common problem of small teams working on medium or short term projects, often under intense schedule pressure, has only recently begun to be addressed, notably by work on the Extreme Programming [3] model. In this setting, it is perhaps both easier and more necessary to adopt a wider variety of more “lightweight” approaches to software process.

Many of the process elements used in the ICPC process are applicable to this sort of development situation. But more importantly, the identification and selection of appropriate process elements, their combination into a sensible development process, and the constant re-evaluation of this process, are efforts well worth pursuing in small-scale projects.

Much of the process described in this document has been codified and published online (see **Availability** below). In the future, the process will be further tuned. In particular, a greater emphasis on earlier practice and training will be necessary. One interesting unexplored variable is the relationship between individual preparation and team preparation effort. The former has been largely neglected at PSU to date, and there is some evidence to indicate that the students are now largely skills-limited rather than process-limited. Given stronger attention to skills development, better preparation, and the vagaries of fortune, there is reason to hope that this model for ICPC programming will raise team performance even higher in the future.

In addition, as opportunities arise for exploring this sort of approach in industrial settings, they will be eagerly taken advantage of. A close personal friend, who has written hundreds of thousands of lines of industrial-quality code on a solo basis, was recently asked for an opinion on software engineering. The response was telling: “I wish there was some.” As lightweight software process elements begin to percolate into this sort of development, perhaps this wish can begin to come true for a larger part of the development community.

## **Acknowledgements**

The author wishes to thank the 1999 and 2000 PSU ACM ICPC teams for their support, encouragement, and tolerance during the development of the materials described here. They are all terrific people, and should be congratulated for their hard work, dedication, and talent. The support and encouragement of the faculty, students and staff of the PSU Computer Science Department and College of Engineering and Computer Science is gratefully acknowledged. In particular the encouragement and advice of Warren Harrison and Dick Hamlet in the preparation of this document have helped to make this work a reality.

## **Availability**

The process model and training materials described are available to the Portland State University ACM Programming Team at <http://bart.cs.pdx.edu/acm-prog>. Those interested in sharing in the development and use of these materials are encouraged to visit the web site, and to contact the author at [bart@cs.pdx.edu](mailto:bart@cs.pdx.edu) to obtain access to the protected portion of the content.

## References

- [1] The 2001 ACM International Collegiate Programming Contest. Web document <http://acm.baylor.edu/acmicpc/> accessed June 23, 2001 07:19 UTC.
- [2] Donald Bagert and Barbara Boucher Owens. Organizing a team for the ACM programming contest. In *Proceedings of the 26th Technical Symposium on Computer Science Education*, volume 27 of *SIGCSE Bulletin*, page 402. ACM Press, March 1995.
- [3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
- [4] Frederick P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, April 1987. Reprinted in [5].
- [5] F. P. Brooks, Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, second edition, July 1995.
- [6] Raymond Klefstad. The UCI Programming Contest Handbook. Web document <http://www.ics.uci.edu/~klefstad/contest/> accessed June 22, 2001 21:02 UTC.
- [7] Glenford J. Myers. *The Art Of Software Testing*. John Wiley & Sons, 1979.