

Trends & Review

May 4, 2005

I. Trends

- o Easy ones: Internet everything, Java
- o Server programming
 - High availability, stable memory footprint
 - Many parallel connections
 - Client-Server communication: TCP, RMI, HTTP, ...
 - User tracking
 - Security
- o Dynamic Content: merging of HTML and programming
 - Mixture of graphics/art/layout with computation of the content
 - Database integration
 - Personalization & log analysis
- o Reuse of big objects:
 - COM, DCOM, Corba
 - Third party libraries
 - Servers as part of applications
 - Combining data from many sources, each with distinct access methods
- o Embedded systems
 - Software for small devices, appliances, phones, and yes, lightbulbs
- o Mobile computing: access anywhere, anytime, customized to you and your device
- o Faster time to market:
 - Parallel development teams
 - Lower quality products
 - More need for process and defect prevention (by tools, discipline, etc.)
 - More need for technical managers (to make the crunch-time tradeoffs)
- o More room for contribution/innovation than ever before
 - Popular sites can have a million users
 - “Bazaar” model can move your code to a million users
 - Applets can also reach millions

Trends & Review

II. Coding Lessons

- o Early integration always wins
- o The single-location rule: all documents, specs, and code fragments should be in exactly one place. This ensures consistency and avoids propagating defects via copy-and-paste (cut and paste is fine).
- o API's and class definitions are CONTRACTS. Therefore, minimize what you promise and leave your options open.
 - Subclasses inherit the contracts of *all* ancestors, not just the prototypes! (Same with interfaces in Java)
 - Changing a class requires changing all subclasses as well!
 - *If you change the spec, change the prototype.* This forces rewriting (slightly) all derived classes and clients of the the prototype. However, this is good, since they are likely ALL WRONG anyway, and this forces to you verify that they meet the new spec.
- o Use preconditions and invariants. They are a simple precise form of documentation, that can be validated explicitly during debugging (with asserts).
- o *eliminate all warnings* in the steady state. (Use gcc -Wall for Unix) Otherwise you'll ignore the important ones
- o Three issues must be treated differently: availability, security, and safety
 - Need 100% solution, not a best effort solution
 - Focus on a half-page simple argument about why the system has these properties
 - Adding complexity weakens all three severely -- the half-page argument must be independent from the complexity of the individual modules. This means that these overall properties can't really depend on the correctness on any complicated module. (Hardware interlocks are an example of a simple module.)
- o Pair up alloc/dealloc (or use Java)
- o Code review works better than anything else to remove defects
- o Regression testing is a must for serious software.

III. Big Picture

- o Take the initiative, solve the problem (don't be loner hero, though)
- o Team stability and productivity is the single biggest factor in productivity
- o Change is coming: focus on maintainability even over correctness
- o Respond to your user! (but educate them first -- 80% of Microsoft's "new" feature requests are already in the product)
- o Have phase reviews: each one is a no/go decision; a "go" comes with more resources to complete the next phase. Kill projects early not late. Invest management time up front, where it will make a difference, not at the end, when it's too hard to turn the ship.
- o Your job is fundamentally to create intellectual property.