

# OOPSLA 2007 "No Silver Bullet" workshop position paper

## Mark Sawers

Fred Brooks "No Silver Bullet" has withstood the test of time. The problem statement and conclusions are, in large part, still true today. The quantity of software intensive systems has certainly grown, and the number of engineers as well, but product quality lags this growth. Software Engineering (SE) is an immature discipline. SE has made strides, if you consider the incredible pervasiveness of technology today (e.g. cell phones, the Web). The world has become more connected, faster and flatter. These advances would not be possible without advances in SE; however, SE still faces major challenges in estimation, productivity, quality and flexibility.

SE is more art than engineering, though the trend is towards engineering. The critical problem-solution pairs, or at least the first principles, have not been discovered, published, refined and codified. This is challenging because no two software systems will ever be identical. Differing problem domain and deployment contexts call for differing software solutions. But my hope is that if we have a set of repeatable, codified solutions--a practitioner playbook--we can build any new system on schedule that meets the customer need with the desired quality.

Brooks describes the essential difficulties of software engineering in four areas: complexity, conformity, changeability and invisibility. On the whole, I agree with the characterization, except with conformity. Yes it exists, but this is not a unique quality. Every other engineering discipline must manage constraints, be they physical, contextual, legal, monetary, etc.

From these essential properties, I derive five challenges unique to SE: communication, abstraction, flexibility, integration and verification.

### Communication

On the front end, the creation of a new system requires a good relationship between the customer and the provider, as they are inventing something new together. Customers and engineers need to communicate needs, desires and constraints effectively. They must define the problem in detail and conceive of a technology solution, often iteratively. Furthermore, communication within an engineering team is a challenge, not only for plans and status, but also in representing the requirements, the design and the implementation.

### Abstraction

Because of the invisibility of software, solution representation is also challenging. We must deal in abstractions, rather than concrete, geometric realities. We use multiple, multi-layered representations for multiple audiences--from customers, to developers, to machines. The forms differ at design time versus run time, and in depicting design versus implementation.

### Flexibility

Because requirements are explored iteratively, rather than dictated, and because they often change once they are "discovered", software designs must be flexible. They must grow in expected and unexpected ways (extensibility and adaptability). I would posit that the key to flexibility is modularity, the organization of software into loosely coupled, highly cohesive modules (components). Defining, much less achieving, the right amount of modularization for the current needs of both the customer is currently more art than science.

### Integration

Most software-intensive systems are created with multiple individuals who create multiple software modules. Integrating these components together presents a fundamental challenge. It is where most architectures, project management and implementation skills are truly tested. The key areas of focus are communication (plans, relationships, specifications), practice (layered testing, good design, coordinators and testers) and tools (build, verification).

## Verification

Software systems of any interest have lots of features, lots of components (modules) and possibly many deployment configurations and use contexts. In some systems it is infeasible to test all scenarios or variations of use. Balancing the cost of testing (time and money) versus quality requirements becomes a game of risk management. Not only must you choose the scenarios to test but you must also choose how frequently to execute them. You can execute them after every software change or do targeted testing, taking calculated risks.

I do not believe there will ever be a silver bullet, as the problem with SE is multi-faceted. Multiple vampires call for a whole magazine of bullets. We need improvements in education, process, practice and tools. I hold out the same hopes as Brooks, but I would add to his list the following three practices and one tool:

## Modeling

Modeling, particularly in UML, has provided a huge leap forward in addressing communication and abstraction challenges. It has created a richer dialog between customers, designers, implementers and testers, allowing them to represent software structures, behavior and business process at different layers of abstraction to depict various system aspects. Furthermore, modeling tools promise productivity increases. They are getting better at transforming design concepts into implementation, and vice versa.

## Iterative Methods

Agile processes have supplanted waterfall and classic SDLC as the standard process at many organizations, and at least influenced processes in many others. They dispense with unnecessary ceremony, dropping the low-value artifacts and activities, encouraging direct partnerships with customers, and early and frequent deliveries of working software. This requires high-functioning teams, frequent builds (e.g. daily), continuous integrations and test automation. Design is iteratively evolved, though architectural risks are addressed up front. These approaches aim to deliver better software faster.

## Test Automation

Multi-layered test approaches (unit, component, subsystem, system) have been around for decades, but the tools, and probably the discipline, has not. At the unit level, frameworks like XUnit, have exploded in popularity. Benefits include improved correctness, design (better interfaces, modularization), documentation and maintainability (a safety net for maintainers). At the system level, functional/GUI and performance test tools, have offered huge productivity gains with industrial-strength scripting environments. Organizations are finding value in their test harness investments. With ever evolving software, test cycles must be repeated. The promise of test automation is more agility--the ability to turn out a higher quality product as well as make changes faster.

## Software Reuse

The vast majority of software projects are built on layers and layers of acquired software: operating system, GUI framework, server containers, databases, IDEs, build tools, profilers, Web frameworks, communication libraries and so on. Typically only the top layers are custom code. This infrastructure is both COTS and open source. It is increasingly easy to share software on sites, like sourceforge.net, and choose from a variety of licensing models. We're a long way from plug and play, but the trend in software implementation is towards more integration and configuration away from programming. The vast majority of reuse is addresses horizontal concerns; however the market for vertically-oriented systems, such ERP, CRM and sales management systems, is also growing.

Mark Sawers

Software Architect, IPC Systems