

Blueprint for a Software Factory

Will industrializing your component development create better apps?

By Michael Morris
May 2008

There is an ongoing debate within the software development community on whether the value of software reuse is ever fully realized and what type of development methodology can make it happen.

Imagine that software projects are built out of cement blocks and the act of putting those blocks together in different ways is what makes each software project unique. Now, replace the physical cement block with a virtual software block called a software component. Next, put all the focus on how to create a repeatable process to build those software components-similar to a cement block manufacturing plant modifying and improving their production processes-with, in almost all cases, no regard to how the blocks will ultimately be used.

Assembly Line

Building a software factory for reuse requires that we switch our mindset from a traditional software project-where a limited number of resources act as a jack of all trades and master of one to none, such as product manager, designer, architect, developer, quality assurance, system administrator and network specialist-to something more efficient. In the software factory, the process for building a component is broken down into fundamental tasks that can be performed independently, by resources that are skilled and trained in that specific task. Imagine a step-by-step process that's modeled after an assembly line, where each component goes through a series of distinct and rigorous actions. Here's how to get started:

1. Define the functional behavior of a component in a way that abstracts the larger system from the component requirements. This creates a very necessary barrier between the resources working on that component and the overall project for which it's initially being built.
2. Determine the technical constraints of that component in measurable and specific terms such as speed, throughput, environment and failure behavior. The use of .NET greatly simplifies the environment issue because there's no need to worry about the details of the operating system or hardware, just the version of the .NET Common Language Runtime.
3. Have experts design that component in a set of clear blueprints using Unified Modeling Language (UML) deliverables. You can enforce the creation of use-case diagrams, class diagrams and sequence diagrams for this task.
4. Review the design with a panel of peers and expert designers to provide measurable feedback and direction on the design. This provides the quality control and best practices required for the success of the factory.
5. Move to development and have your team members build the component to spec. This step is merely following directions and working within a controlled environment, where creativity is limited.
6. Most importantly, evaluate, measure and complete the project. Measure everything including the resources involved, cost, time, code quality, test coverage, bugs, enhancements, deployments, inquiries and return.

The key to this six-step process is that everything must be tracked with metrics, so the process can be properly evaluated and continuously improved. It can take several iterations of the factory to get it working at a predictable level.

Lean and Clean

Once the software factory is up, running and producing high-quality products, the factory is ready for integration into your project methodology. Most people inquire if waterfall or agile methodologies are used. The answer is yes and no. Both methodologies need to be altered to take advantage of component-based development and maximum reuse. For example, a component-based waterfall methodology is traditionally thought of as slow and laborious. If you can merge that with a parallelized factory approach for development, you've just given the waterfall methodology a hyper-boost, reducing delivery time.



For example, in working with AOL, the functional requirements doubled in size, but the development time was not affected-not including requirements gathering and architecture. This is a very powerful incentive for teams to over-deliver and outperform, where most expect a linear increase in time with each additional use-case or function-point.

For the component-based agile methodology, it's important to have your team members focus on and understand the value of existing code. The factory has to be used in two primary ways, depending on whether the components already exist or require development through the factory. Existing components are a pretty clear value because they can increase the functionality you're able to implement in a given sprint. New components are a little trickier, as they take nearly as long as the sprint. In this case you can complete a sprint and leave a place for the component to be inserted. If the delivery and quality of your factory is predictable enough, you should rarely have issues with the components not fitting into their proper place within the application. A metric to track this is bugs per thousand lines of code, or .72 defects per KLOC.

Today, we're in a position to see software factories do the same things accomplished by manufacturing plants in the 1800s: drive down cost, increase quality and improve speed to market. I'm a strong supporter of this software factory approach, but will warn adopters in advance that it requires a commitment to process, measurement and not cutting corners-often a difficult trio to enforce on existing development groups. Good luck!

Michael Morris is a senior vice president at TopCoder Inc.

