Does Your Code Hurt? Help the Team to write it Better Then

It seems to me that too many developers have the perception that, at the very end of day, bad code in a project and especially a large project doesn't hurt that much. And, yes, if you count the number of projects which reportedly failed for code-related issues, well, I have to agree that the number is not huge. At any rate, I don't think you have to wait for a true disaster to lose good money on a software project.
As a lead developer (solution architect or whatever is the title on the business card), what can you do to help the team writing better code?

Seven Virtues of Software Development

I dare say that successful projects are based on two factors: management that knows about leadership and development that knows about code quality. In summary, here's a list of seven virtues that may really take a team to the paradise of software.

#1 Have Tools Assist Coding

As far as coding is concerned, there doesn't have to be a time for developers to write just code and a (later) time to fix and clean it up. This second pass never happens. Better code should come at first shot. A significant help comes from code assistant tools. Typically integrated into the IDE, these tools simplify common development tasks making developers proceed faster and with the great chance of writing really better code. At worst, code is written faster possibly leaving some time for a second pass.
Services like auto-completion, tips on idiomatic design (that is, writing code in accordance to the language or framework suggested idiom), code inspections, predefined snippets associated with keystrokes, and predefined and customizable templates available are all practices that speed up development and ensure consistency and possibly better and cleaner code.
Code assistant tools make development a sustainable effort and vastly improve the quality of the code you write with a couple of extra clicks. A code assistant tool can catch duplicated and unused code, can make refactoring a pleasant experience, simplify navigation and inspection and force patterns. More than everything else, tools help adopting an appropriate naming convention making renaming and method refactoring a breeze.
ReSharper is the most popular of the code assistant tools available out there. For more information you can visit *http://www.jetbrains.com/resharper*. Other analogous tools are CodeRush from DevExpress (*http://www.devexpress.com/Products/CodeRush*) and JustCode from Telerik (*http://www.telerik.com/products/justcode.aspx*).

#2 Tell People When Their Code Is Bad

Suppose you noticed that some people on the team are writing bad code. How would you go telling them about it?
There are psychological aspects involved here. You don't want to be sharp and you don't want to hurt anybody; at the same time, you don't even want that other people's work can turn on you at some point.
The best way to focus attention on some code you don't like is subtly asking why it was written the way it is. You may find out more about the motivation, whether misinformation, bad attitude, limited skills or perhaps constraints you just didn't know.

It is extremely important that you never judge your coding alternative better without clear evidence. So you can simply look curious and interested in the real motivation for such a code and willing to know more as you just would have coded it differently.

#3 Make Everyone a Better Developer

I like to summarize the golden rule of having teams writing good code as below:

Address the code, not the coder. But try to improve the code through the coder.

You can have any piece of bad code fixed in some way. When this happens, however, you don't want to blame the coder; you do want, instead, to help the coder to improve his way of doing things. If you're able to do this, you get benefits in at least two areas. You get a better developer on the team and you have possibly a happier and more motivated developer on the team. You made the developer feel as close to the hero status as never before because he now has the perception of having done its job in the best possible way.

#4 Inspect Before You Check-In Code

You can have the best coding standards ever in your company but how would you enforce them? Trusting developers is nice but verifying is probably more effective. Peer programming and regular design reviews are a concrete way to check the health of codebase. In a typical design review, you can manage to take some sample code and discuss it openly and collectively. The code can be a real snippet from the project--that some of the participants actually wrote--or, to avoid emotional involvement, it could even be an aptly written piece of code that just illustrates the points you want to make.

To enforce coding standards you can also consider applying check-in policies to your source control system, whether TFS, TeamCity, or others. Can the process be automated in some way?

Today nearly any source code management tools offer ways to exercise control over files being checked in. For example, TFS has gated check-ins. A gated check-in is essentially a check-in subject to rules.

In other words, files being checked in will be accepted into the system only if they comply with established rules. When you choose to create a gated check-in, TFS invites you to indicate an existing build script to use. Only if the build completes successfully files will be checked in.

In TFS, a build is just an MSBuild script and can be customized with a variety of tasks. TFS comes with a number of predefined tasks that can be integrated. For example, you find a code analysis (formerly FxCop) tasks and a task for running selected test lists. As an MSBuild task is nothing more than a registered component that implements a contracted interface, you can create new tasks yourself to add your own validation rules.

It is worth noting that one of the aforementioned code assistant tools--JetBrains' ReSharper--in its latest edition features a set of free command line tools that can be used in a custom MSBuild task to detect duplicated code and apply typical inspections, including custom inspections against custom templates you may have defined. Interestingly, you don't even need a ReSharper license to use command line tools. For more information on ReSharper command line tools, have a look at *http://www.jetbrains.com/resharper/features/command-line.html*.

#5 Happy the Project that Doesn't Need Heroes

We developers tend to have a super-ego and at least in their most secret dreams wish they could work 80+ hours a week to save the project, keep customers happy and be true heroes to the eyes of management and fellow developers.

Sometimes deadlines are unfair since the beginning. In some other cases, deadlines prove wrong along the way. When this happens, it's emotionally easier to just acquiesce, but that won't really help out and, more, brings in the need of heroes. Communicating and making troubles clear to

anybody is a matter of transparency but also an effective way to regain a bit more of control and reduce pressure.
In software, we experience pressure because of looming deadlines or lack of skills. Both situations can be addressed properly if communicated timely. There should be no need of software heroes; and while I’ve been hero myself quite a few times I think I have learned that heroism is an exceptional situation. And in software exceptions are mostly something to avoid.

#6 Encourage Practice

Why on earth professional players of nearly any sports are used to practice every day for hours? Is there really some analogy between developers and professional players? It depends.
One way to look at it is that developers practice every day at work and don’t compete with other developers. Based on this, one can conclude that there's no analogy and subsequently no point in practicing.
Another way to look at it is that players exercise basic movements very frequently so that they can repeat them automatically. Going regularly back to the foundation of object-orientation, design patterns, coding strategies, certain areas of the API helps keeping knowledge in the primary cache and more quickly accessible.

#7 Continuous Change Is Part of the Deal

A software project begins following an idea or a rather vague business idea. Most software projects are just like moving targets and requirements are what move the target around. Continuous change is rather news than novelty. As obvious as it may sound, mindset is the only effective way to fight it. To be effective, every developer should constantly be ready to refactor.
Refactoring is one of those things that are hardly perceived as bringing value to the project. Bad luck that failing on refactoring takes value away.

Bad Code Is Really More Expensive than Good Code

In the context of a project with a significant lifespan and business impact, writing bad code ends up being more expensive than writing good code. As simple as it may sound, a project dies of bad code when the cost of working with code—that is, creating, testing and maintaining it—exceeds the cost that the business model can bear. By the same token, no projects would fail for code-related issues if companies can manage to keep the cost of code extremely low.

And this is just the sore point.

Sometimes managers boil it down to just cutting development costs, hire cheap developers and ask/order to cut off tests and documentation. They lower the cost of producing code and take the project home. Unfortunately, producing *code-that-just-works* is only one aspect of the problem. In modern times, requirements change very frequently, complexity grows and, worse yet, is often fully understood only as you go. In this scenario, producing code is only one item in the total bill of costs. Code maintenance and evolution is the biggest item.
As the good architect knows very well, one can address code maintainability only through well written code, good understanding of software principles and language features, well applied patterns and practices, testability. This makes coding more expensive than producing *code-that-just-works,* but far cheaper than maintaining and evolving *code-that-just-works*.