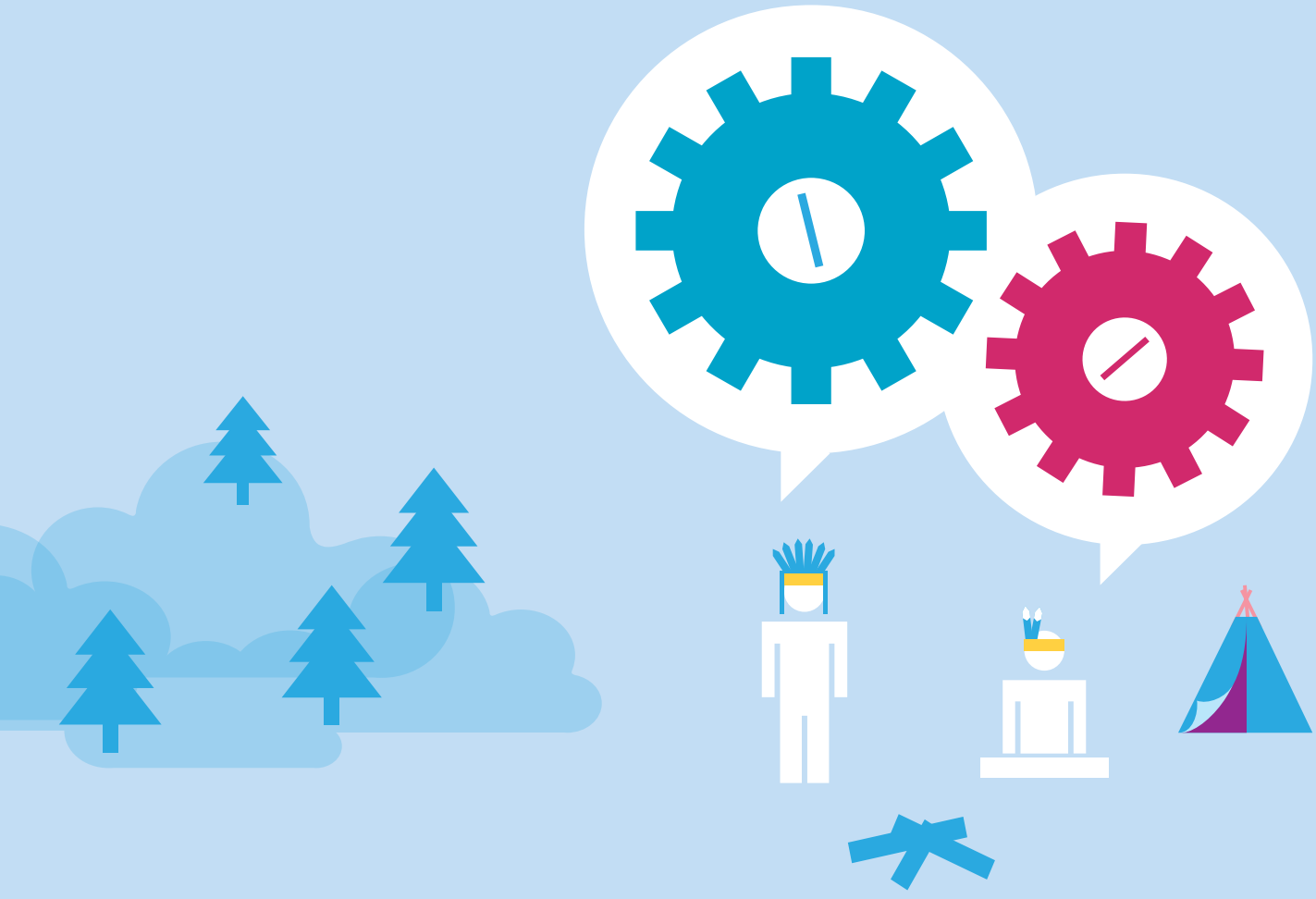
→ THE DEVELOPER/TESTER

# GUIDE TO BETTER COLLABORATION

Or why developers & testers should bury the hatchet?



Telerik
**TEST STUDIO**

www.telerik.com/test-studio www.twitter.com/teleriktesting

# Contents Introduction:


INTRODUCTION: WHY COLLABORATE?.....2

THINGS TESTERS CAN LEARN FROM DEVELOPERS.....3

- Backing APIs.....3
- Configuration / Switches.....5
- Craftsmanship and Code Smells.....6
- Testable User Interfaces.....10

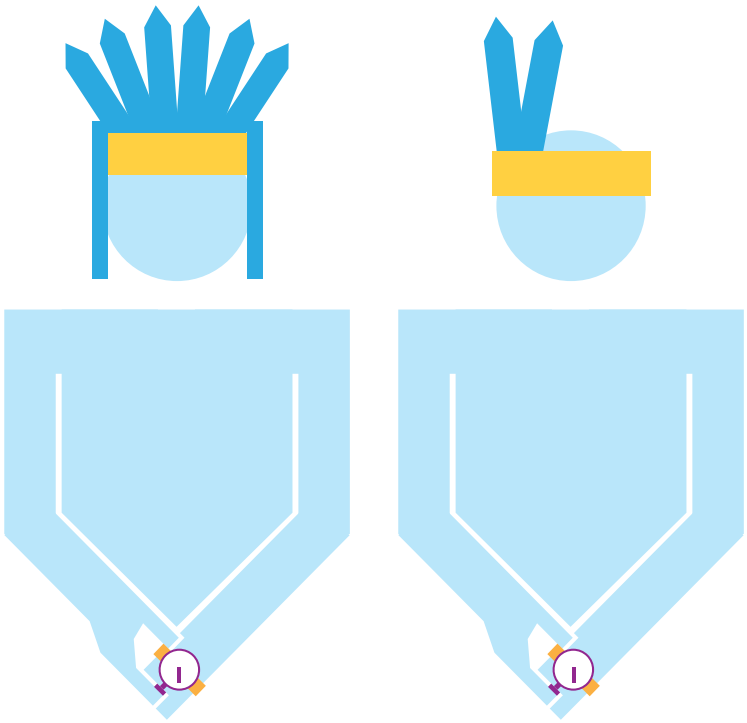
THINGS DEVELOPERS CAN LEARN FROM TESTERS.....13

- Good Test Case Design.....13
- Improve Error Handling and Sad Path Coverage.....13
- Validate value of work item.....15

CONCLUSION: WHY COLLABORA TE?.....16

COLLABORATION: IT PAYS OFF.....16

# Introduction: Why Collaborate?



Why should testers and developers collaborate?

It's a perfectly legitimate question, particularly to those who've been in the software industry for a number of years and have seen the coming and going of any number of buzzword fads.

However, collaboration among members of a team producing software isn't just a fad. The IT industry is finally moving away from stovepiped, separated groups to a much healthier, more productive whole team environment. Case studies and experience reports are increasingly confirming the value of this transformation.

This guide focuses on one aspect of whole team interaction: collaboration between developers and testers. Both roles bring tremendous skills and experience to a team. Having the two work together often results in a marked improvement in the quality of work, and a noticeable decrease in waste and revision.

We'll examine how each role helps the other to look at software development in new ways.

# Things Testers can learn from Developers

Good developers bring solid design, engineering, and craftsmanship expertise to a team. Good testers should view partnering with developers as an extraordinary opportunity to expand their skills, and do so whenever possible. Testers can adopt many concepts from developers to make their test suites more valuable, maintainable, and powerful.

## Backing APIs

Backing APIs, sometimes called test support infrastructure, are critical to a flexible, powerful, and maintainable automation suite. Backing APIs let you leverage your system's internal functionality to handle things like configuration, data creation and cleanup, or test oracles. These sorts of actions can sometimes be performed by UI automation; however, they're better left to faster, more flexible methods such as web service endpoints, internal APIs, or stored procedures.

Many testers are often hesitant to try this approach themselves, since few testers are comfortable writing database accessors, web service calls, or system call invocations. In these cases, reaching out to developers for help makes perfect sense.

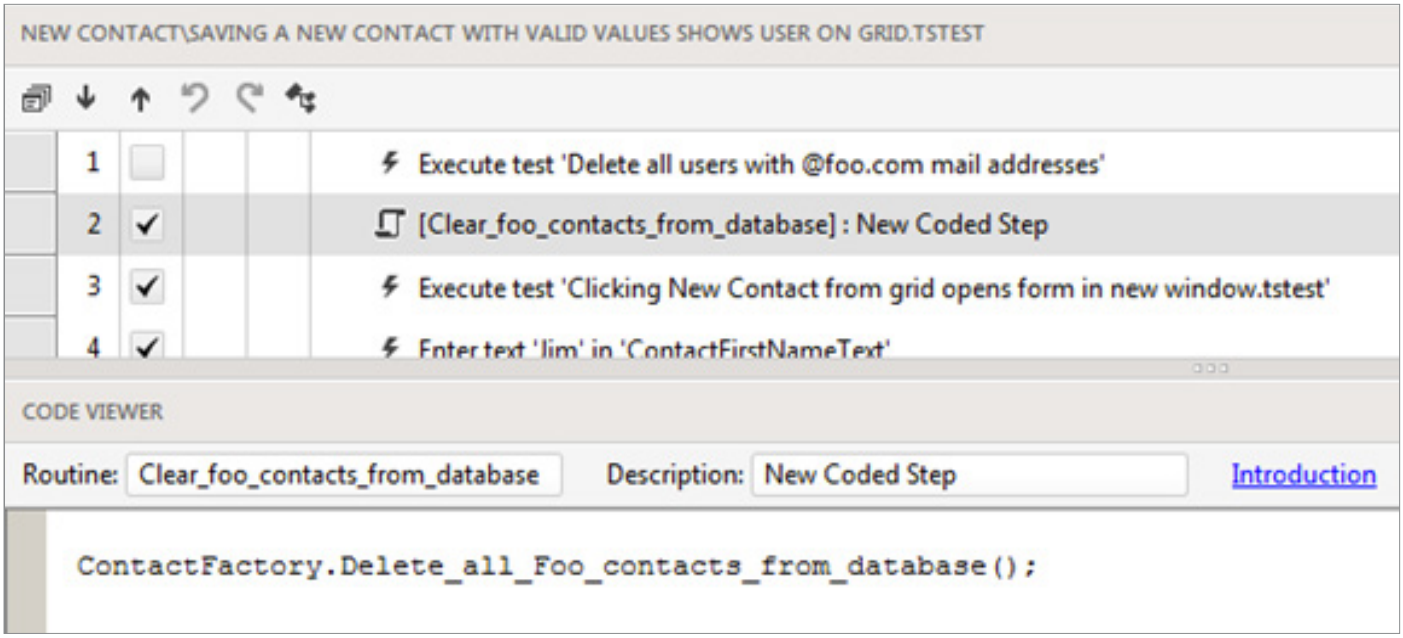
For instance, let's look at a test that creates a user in a system:

| | | |
|---|---|---|
| 1 | ☐ | Execute test 'Delete all users with @foo.com mail addresses' |
| 2 | ☑ | [Clear_foo_contacts_from_database] : New Coded Step |
| 3 | ☑ | Execute test 'Clicking New Contact from grid opens form in new window.tstest' |
| 4 | ☑ | Enter text 'Jim' in 'ContactFirstNameText' |
| 5 | ☑ | Enter text 'Haswell' in 'ContactLastNameText' |
| 6 | ☑ | Enter text 'jh@foo.com' in 'ContactEmailEmail' |
| 7 | ☑ | Enter text 'http://mypage' in 'ContactLinkedinProfileText' |
| 8 | ☑ | Check 'ContactGovtContractCheckBox' to be 'True' |
| 9 | ☑ | Check 'ContactDodCheckBox' to be 'True' |
| 10 | ☑ | Desktop command: Drag & Drop NeutralIconImage to Contact Type Drop Target |
| 11 | ☑ | Click 'CommitSubmit' |
| 12 | ☑ | Verify 'TextContent' 'Contains' 'jh@foo.com' on 'JhFooComTableCell' |
| 13 | ☑ | Verify 'TextContent' 'Contains' 'Jim' on 'JimTableCell' |
| 14 | ☑ | Verify 'TextContent' 'Contains' 'Haswell' on 'HaswellTableCell' |
| 15 | ☑ | Verify 'TextContent' 'Contains' 'http://mypage' on 'HttpMypageTableCell' |
| 16 | ☑ | Verify attribute 'alt' has 'Same' value of 'Neutral' on 'NeutralImage' |
| 17 | ☑ | Verify attribute 'src' has 'Same' value of '/assets/NEUTRAL.png' on 'NeutralImage' |
| 18 | ☑ | Oracle: Validate newly created user is in database |
| 19 | ☑ | Close pop-up window : '/contacts' |

The purpose of this test is to validate whether a properly created user is persisted in the system's database. We want to avoid any error handling around duplicate user creation—tests shouldn't deal with error handling, they should focus on checking the validity of the tested slice. We can avoid this problem a couple different ways: ensure we create a unique user each time we run this test, or we could ensure all test users are deleted before we run this test.

Testers could write UI automation scripts to handle this task (start a browser, log on to the system, navigate to the system's administration section, delete any existing test users, e.g.), but that's slow and brittle. Teams are much better off leveraging code-level APIs within the system itself. Step 2 in the figure above does just that, via this bit of code:

Here a developer has created a simple method (Delete_all_Foo_contacts_from_database) on a helper class (ContactFactory) in order to clear out test users. This makes it easy for less-technical testers to get the job done using just enough code, without having to understand either the deep internals of the system or the technical details of invoking a web service. Note that how this method works is hidden from the user of the backing API. This concept—abstraction—is critical in good software design. Abstraction separates what something does from how it is done. The tester doesn't know, or care, if the ContactFactory is calling web services, internal APIs, or a command line utility. This enables the team members maintaining the backing API to switch to the most appropriate approach for the particular operation— and the testers would never have to touch their tests!



## Configuration / Switches

Automation professionals are often asked, "How do we automate CAPTCHA or similar difficult third-party features and tools?" The correct answer is nearly always, "Don't."

CAPTCHA is a perfect example of something that should be bypassed or turned off, versus struggled with. The point of an automated registration test shouldn't be checking a third-party bot filter (CAPTCHA), the point should be to ensure a newly registered user actually populates in your database. Futzing around with trying to detect CAPTCHA graphics and work through it is simply a waste of time and doesn't bring value to your automation suite.

Testing sent e-mail is another area fraught with frustration. The last thing testers should ever be doing is writing tests that log on to Gmail in order to validate formatting and content of system-generated mails. Both CAPTCHA and email are perfect examples of collaborating with developers to control system configuration during automated test passes.

There's no reason we shouldn't have separate system configurations for testing and production, as long as we carefully control (and test!) the deployment process to ensure no critical functionality is shut off in our production environments. This approach enables testers, developers and IT team members to work together to change the system to make it more testable within certain constraints. Developers will have to do additional work allowing features like CAPTCHA or mail providers to be swapped out or shut off; however, careful discussion should enable the team to if it makes sense to undertake such an effort.

Exact implementation of the configuration switches will be extremely specific to each system under development; however, here's how one implementation might look for a .NET application hosted under IIS using a web.config file:

```
class Web_config_switches
{
    public void shut_off_captcha()
    {
        change_appSettings_key_value("captchaActive", "false");
    }
    public void turn_on_captcha()
    {
        change_appSettings_key_value("captchaActive", "true");
    }
    private static void change_appSettings_key_value(string key, string value)
    {
        string path_to_config = @"c:\some_dir\web.config";
        Configuration webConfig =
WebConfigurationManager.OpenWebConfiguration(path_to_config);
        webConfig.AppSettings.Settings[key].Value = value;
        webConfig.Save(ConfigurationSaveMode.Modified);
        ConfigurationManager.RefreshSection("appSettings");
    }
}
```

This snippet of code assumes the system under test has a section of its web.config file which includes a captchaActive flag. The system under test would obviously need to support altering CAPTCHA status based on that flag—and details of that implementation are far beyond the scope of this work.

Automated tests could simply reference Web_config_switches.shut_off_captcha() directly from a setup step or test in their tests or lists as appropriate.

Note one significant caveat when working with system switches or changes in configuration: you absolutely must have a set of automated tests that verify the system is correctly configured when deploying to non-test environments. These automated checks must be part of your regular deployment processes, otherwise you risk potentially rolling out your system to production with critical features deactivated. You do not want to be on the receiving end of that call at 2:42 AM.

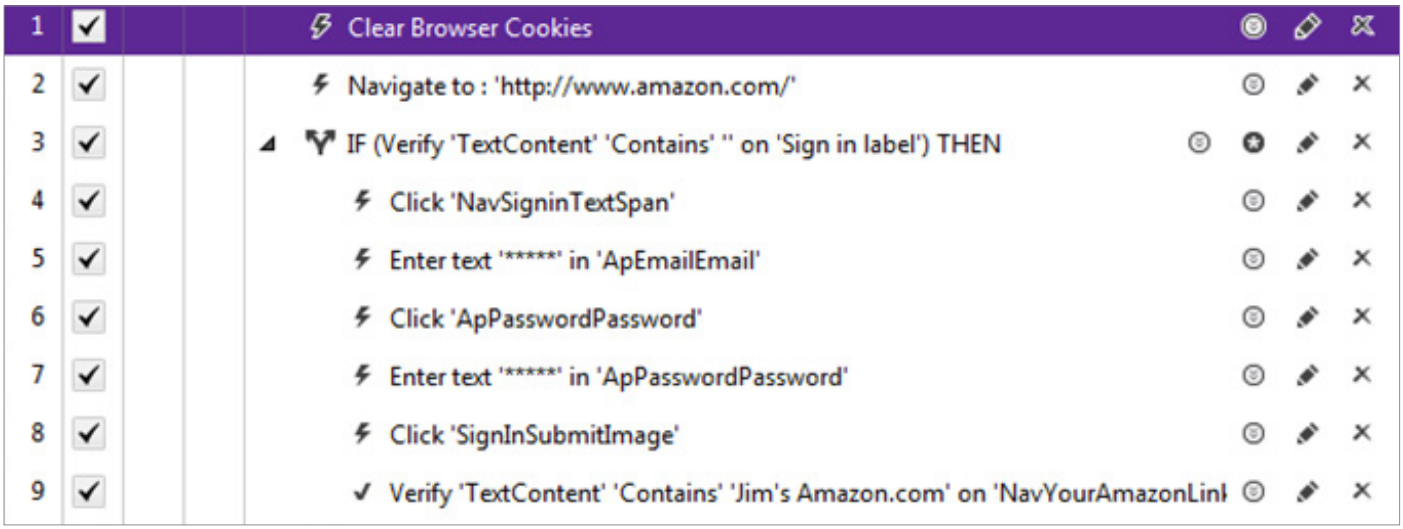
## Craftsmanship and Code Smells

Software craftsmanship and software engineering disciplines have a direct correlation to good testing. The software craftsmanship movement brings a sense of pride in one's work, and frames that in the mindset of carefully learning good practices along an entire career of work. Software engineering contributes concrete metrics and practices to the show in a great compliment to the craftsmanship movement.

Good testers can take several principles to heart from both domains. Good developers know to look for "code smells," clear indications a section of code is too complex, potentially a maintenance nightmare, or flat out wrong. (The term "code smell" was apparently first coined by Kent Beck and Martin Fowler as part of the work for Fowler's seminal Refactoring: Improving the Design of Existing Code.)

## Code Smell: Complexity

Code smells come in several areas. First off would be overly complex code. Nested 'IF' statements in code have long been recognized as a direct contributor to overly complex, hard-to-understand, bad code. See Wikipedia's section on Cyclomatic Complexity as a starting point. The same concept goes for tests as well, as the following image illustrates:



## Code Smell: Mixed Concerns

IF statements are bad practice that they often mix several concerns. The test above checks at least three different test flows (tires, videos, computer supplies), plus logging on if needed. Mixed concerns indicate the test case isn't well-focused—it's working on too many things at once. A failure in one section of this will likely mask potential failures in other areas.

Finally, mixing numerous concerns in one test case makes the case harder to maintain. How do you remember where to find the section of your test suite that focuses on checking videos if you have five, ten, or more scenarios mixed in each test case (file)?

In the software engineering/craftsmanship domains, mixed concerns are often referred to as violations of the Single Responsibility Principle. SRP means that one class or method should focus on doing one thing and leave other concerns to different classes or methods.

## Code Smell: Duplication

This same test provides a great example of the Don't Repeat Yourself (DRY) principle. DRY helps you avoid maintenance nightmares incurred when functionality is duplicated numerous times throughout a codebase. If one thing in a piece of functionality changes, you'll find yourself having to update that functionality everywhere it occurs—and the odds of missing one instance escalates proportionally to how often it is repeated.

The logon workflow in steps three to nine is a common feature and will likely be duplicated in every test requiring a logon. The impact of this can't be overstated: imagine having to update hundreds or thousands of your tests when (not if!) your logon process changes.

The logon-related steps should be immediately moved to a separate test which can be used as a component in other tests. This way no other tests need to be updated if the logon workflow ever changes.

## Avoiding Smelly Code and Tests

Developers can help testers avoid these situations by sharing their experience and patterns they've picked up through their work. Testers can learn to head off smelly code via good design principles and practices. Developers can also teach testers about refactoring, the process of changing the structure or implementation of software without changing its behavior.

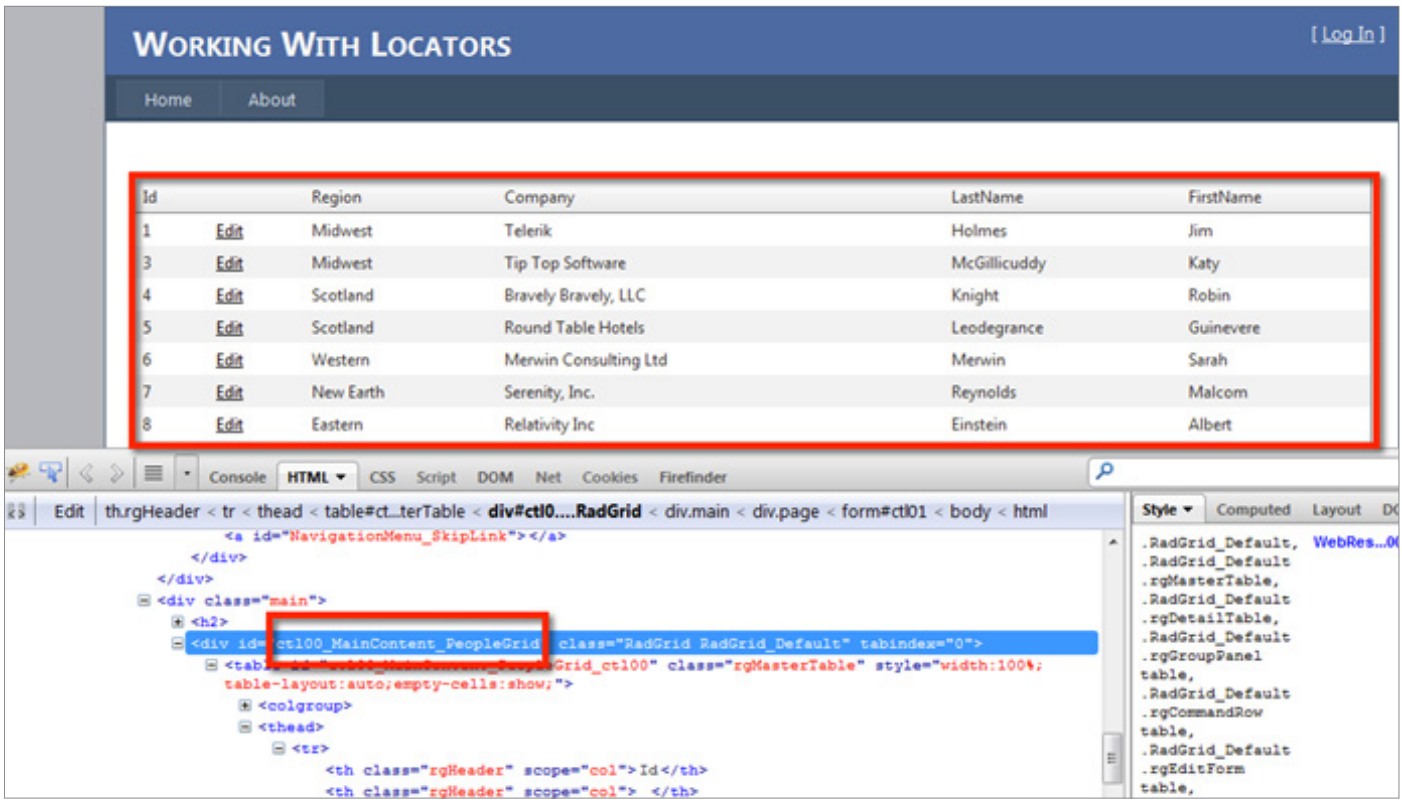
## Testable User Interfaces

Too often, little thought is given to testability at the user interface level. This creates a serious burden on testers who have to create convoluted find logic based on brittle or overly complex XPaths, or rely on fickle conditions such as inner text of target elements. Occasionally testers may mistakenly rely on dynamic IDs for locators.

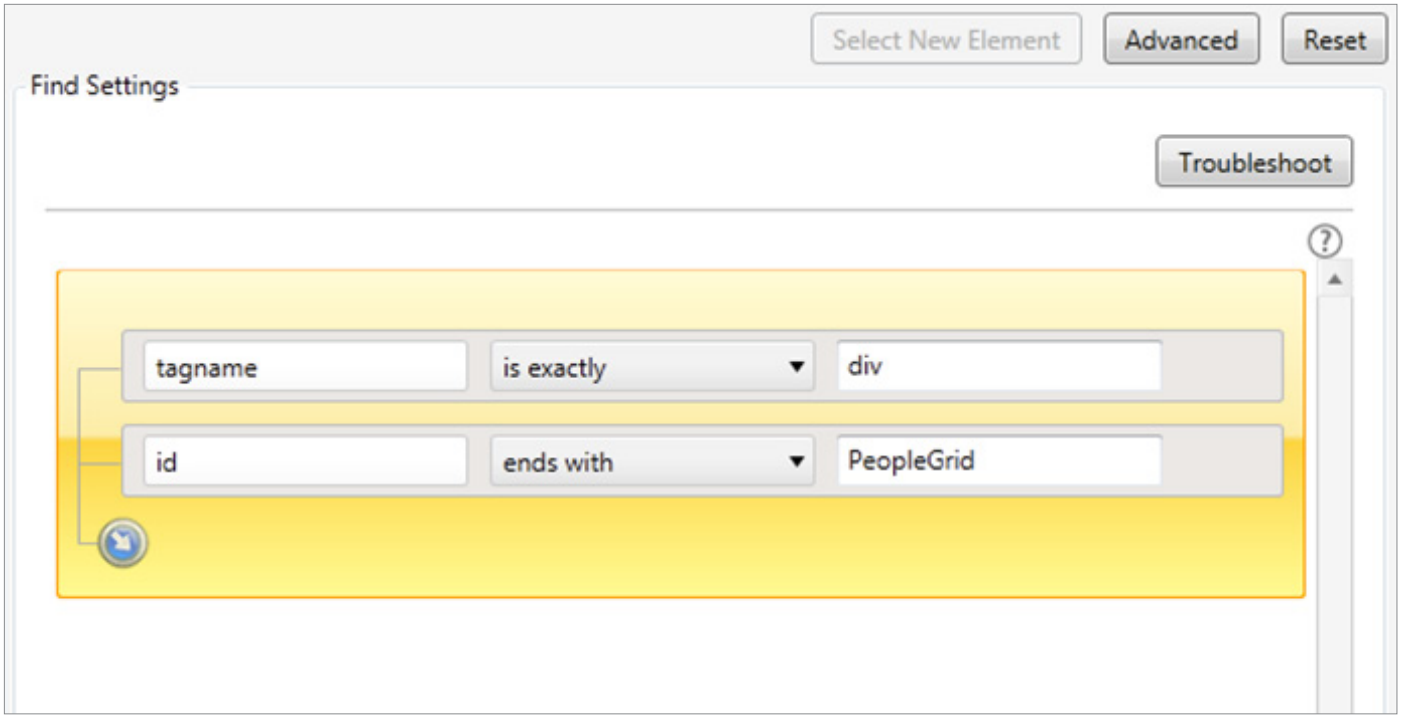
ASP.NET webforms is a particularly egregious offender in this aspect because its generation of ID values is based on the control's position in the overall control hierarchy. An example might be ctl00_SamplesLinks_ctl10_SamplesLink which is dependent on the position of at least two other controls in the hierarchy. This becomes a serious issue when trying to create flexible locators that won't break when controls are added elsewhere in the DOM above the control.

Developers can work with testers to modify the user interface to make it more stable for functional tests. For example, the figure below shows a grid control populated with data. The ID of the grid by default would be ct100_MainContent—totally dependent on the grid never moving from its position in the MainContent div element.

In this example, however, the developer has appended "PeopleGrid" to the grid's ID as shown in the .aspx page's markup below, resulting in the more specific ID "ct100_MainContent_PeopleGrid" as shown below.



This means testers can now use a find expression for this grid using the "EndsWith" form.




These easy steps decouple the grid's find logic from its location on the page and will dramatically increase the test's flexibility when the page layout changes. (Note, that's a when it changes, not if!).

# Things Developers can learn from Testers

No, Grumpy Testers Don't Have Cooties, collaboration between testers and developers isn't just a one-way street. Testers bring a rich, varied view to the team and developers should learn to leverage that in order to improve their own craft.

## Good Test Case Design

Earlier we discussed code smells in software code. As developers become more involved in the whole team approach to software development, they will likely be part of creating test cases—automated, manual, or exploratory/session charters. The same principles of clean code apply to these test cases: it's important to not conflate concerns, create complexity, ensure specificity and validity, etc. Testers can provide critical feedback on these aspects of test case design.

## Improve Error Handling and Sad Path Coverage

It may be a sad stereotype, but too often developers focus on happy paths when designing systems or writing tests. They'll miss critical boundary conditions, and sometimes don't take the broader view on business use or infrastructure issues. Testers can help flesh out better designs for handling likely error conditions around inter-component communication, long-running asynchronous processes, and other architectural or design issues. It doesn't matter that the tester doesn't know how to code up a solution in these instances; it's the tester's domain knowledge and experience that are critical.

Testers can be a great help in pairing sessions whether developers are doing regular development or Test Driven Development as well. Take the following method as an example. It's intended to compute the wages for hourly or salaried workers based on their rate and number of hours worked. Salaried workers get straight time no matter how many hours they work, and hourly workers get time and a half for anything over 40 hours. [NOTE: No, this is NOT production-ready code. This is sample code!]

```
public float ComputeWages(float hours,float rate,bool isHourlyWorker)
{
    float wages = 0; if (hours > 40)
    {
        var overTimeHours = hours - 40;
        if (isHourlyWorker)
        {
            wages += (overTimeHours*1.5f)*rate;
        }
        else
        {
            wages += overTimeHours*rate;
        }
        hours -= overTimeHours;
    }
    wages += hours*rate;
    return wages;
}
```

A developer might come up with a quick set of test cases similar to the following:

| Type Worker | Hours | Rate | Expected |
|---|---|---|---|
| Hourly | 40 | 5 | 200.0 |
| Hourly | 41 | 5 | 207.5 |
| Salary | 41 | 5 | 205.0 |
| Salary | 40 | 5 | 200.0 |

Testers would quickly flesh this out with additional use cases for zero amounts in rate and hours, negative values for rate and hours, and would also likely ask domain-level questions like "How do we handle an hourly worker that switches to salary in the middle of a pay period?" or "What's the maximum amount of hours an employee can work in a pay period?"

This sort of feedback, especially early in a project, can be a tremendous boon to a team as they work to deliver the highest value possible to their customers.

## Validate value of work item

Testers can also provide helpful feedback on basic assumptions made about feature value. Testers often act as customer advocates, and will likely have different insights into customer habits and desires. This can be something as simple as "Customers are very price sensitive and prefer cost as the default sort order, not alphabetic."

More importantly, testers can help validate or disprove the basic value assumption of features before a single line of code is written. "No, we've never had any issues voiced from customers around confusing colors on the test list screen. What we do know is they want better sorting and searching features."

This sort of collaboration can help head off wasted time creating features, and help the team focus on much more valuable, productive work.

# Conclusion: Why Collaborate?

## Collaboration pays off.

As a tester, collaboration with developers may not always be easy, but in the long run you'll be happy you made the effort. Your tests will be more maintainable and easier to write, and you'll be delivering better software to your customers.

[Try Test Studio for free.]