**5** ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

**5.1** Модульное тестирование

Для постоянного контроля качества в процессе разработки системы широко используется модульное тестирование – процесс, позволяющий проверить модули исходного кода программы в процессе разработки. Смысл этого подхода заключается в том, чтобы писать тесты для каждого нетривиального метода исходного кода. Это позволяет достаточно легко проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже написанных и оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны. Модульное тестирование позволяет в дальнейшем проводить изменение уже существующего кода в целях его улучшения, будучи уверенным, что изменяемый модуль и модули, его использующие, по-прежнему работают корректно. Так же модульный тест можно рассматривать и в качестве документирования использования классов: описывая примеры использования класса и его методов, модульные тесты дают понять разработчикам, как правильно использовать класс.

В среде разработки Visual Studio 2013, в которой разрабатывался данный проект, встроены средства модульного тестирования. Для написания модульного теста для выбранного класса, необходимо создать класс, который будет инкапсулировать в себе выбранный класс и помеченный атрибутом [TestClass]. Тестовые методы в таком случае будут помечены с помощью атрибута [TestMethod]. Таким образом, в общем случае модульный тест имеет следующий вид:

[TestClass]

public class QualimetricPredictorTests

{

Team strongTeam;

Team weakTeam;

Match match;

TeamInfo strongTeamInfo;

TeamInfo weakTeamInfo;

QualimetricPredictor qualimetricPredictor;

public QualimetricPredictorTests()

{

strongTeam = new Team()

{

Id = 1, GoalGames = 5, Goals = 15,

DaysOfRest = 14, HomeMatch = 1,

InjuredPlayers = 0, MissingBalls = 0,

Points = 15, Position = 1, ZeroGames = 5

};

weakTeam = new Team()

{

Id = 2, GoalGames = 0, Goals = 0,

DaysOfRest = 1, HomeMatch = 0,

InjuredPlayers = 5, MissingBalls = 15,

Points = 0, Position = 15, ZeroGames = 0

};

match = new Match() { Id = 1, Result = "3" };

strongTeamInfo = new TeamInfo(strongTeam);

weakTeamInfo = new TeamInfo(weakTeam);

qualimetricPredictor = new QualimetricPredictor();

}

[TestMethod]

public void ShouldPredictVictoryIfFirstTeamIsStronger()

{

MatchInfo matchInfo = new MatchInfo(match, \_\_\_\_\_\_\_\_strongTeamInfo, weakTeamInfo);

double [] prediction = \_\_\_\_\_\_\_\_qualimetricPredictor.predict(matchInfo);

Assert.AreEqual(true, (prediction[1] > prediction[2]));

}

[TestMethod]

public void PredictorShouldPredictDrawIfTeamsAreEqual()

{

MatchInfo matchInfo = new MatchInfo(match, \_\_\_\_\_\_\_\_strongTeamInfo, strongTeamInfo);

double[] prediction = \_\_\_\_\_\_\_\_qualimetricPredictor.predict(matchInfo);

Assert.AreEqual(true, (prediction[1] == prediction[2]));

}

[TestMethod]

public void ShouldPredictDefeatIfSecondTeamIsStronger()

{

MatchInfo matchInfo = new MatchInfo(match, weakTeamInfo, \_\_\_\_\_\_\_\_strongTeamInfo);

double[] prediction = \_\_\_\_\_\_\_\_qualimetricPredictor.predict(matchInfo);

Assert.AreEqual(true, (prediction[1] < prediction[2]));

}

}

С помощью аннотации @Before помечается метод, который будет запущен перед каждым тестовым методом. В нем обычно производится инициализация всех нужных для тестирования данных. Аннотацией @After помечается метод, в котором производится освобождение ресурсов после того, как тест отработает. Результат работы тестов на примере тестирования класса QualimetricPredictor в среде разработки VisualStudio 2015 показан на рисунке 5.1.

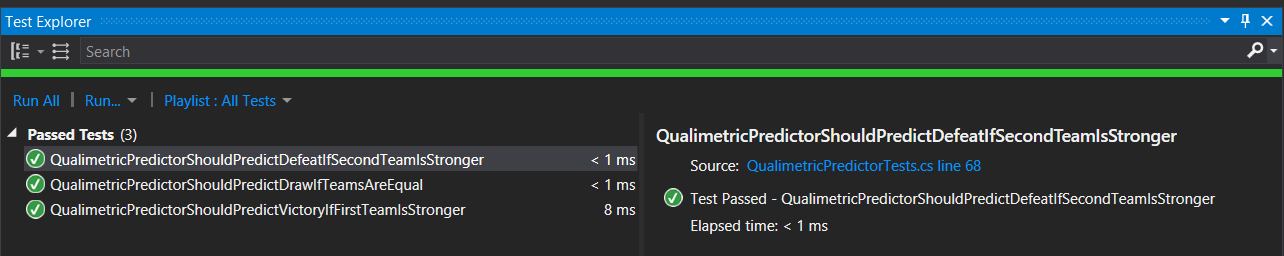


Рисунок 5.1 – Результат тестирования класса QualimetricPredictor

Все классы объектной модели приложения покрыты модульными тестами в среднем на 90%, что обеспечивает низкую вероятность регрессии при дальнейшей разработке и поддержке системы.

**5.2** Обработка исключительных ситуаций

Важной частью методики испытаний системы является корректная обработка исключительных ситуаций. В случае возникновения исключения приложение должно оставаться в рабочем состоянии, а пользователь должен получить уведомление и иметь возможность продолжить свою работу с того места, где возникла исключительная ситуация.

Корректная обработка исключений в .Net веб-приложениях имеет особо важное значение, потому что в случае неправильно обработанного исключения пользователю предоставляется информация о стеке выполнения метода, в котором оно произошло, что является серьезным упущением в безопасности приложения.

Правильная обработка исключительных ситуаций включает в себя следующее:

* исключения должны перехватываться;
* информация об исключениях должна записываться в хранилище данных для возможности ее последующего анализа;
* пользователь не должен видеть полной информации о произошедшей ошибке: стека выполнения метода, деталей исключения и так далее.

Рассмотрим наличие правильной обработки исключений в случае, когда при работе приложения пропало соединение с базой данных, и GamesController не может далее нормально функционировать.

Ожидаемый результат работы программы в подобной исключительной ситуации представляет собой последовательность действий:

* вывести сообщение на графическом интерфейсе о случившейся исключительной ситуации;
* записать информацию об ошибке.

При попытке воспроизвести такое поведение системы, на экран вывелось сообщение о том, что при работе с базой данных произошла ошибка. Данное поведение можно увидеть на рисунке 5.2.

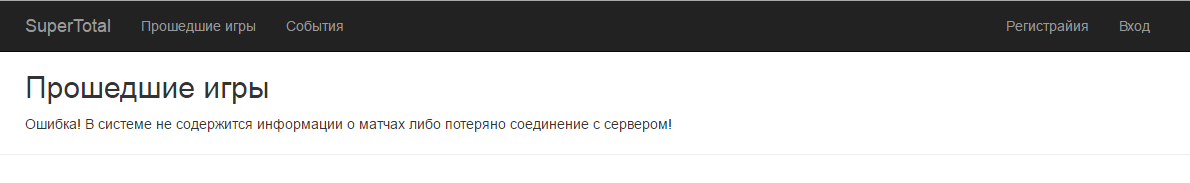


Рисунок 5.2 – Сообщение об ошибке при работе с базой данных

Также все поля для ввода данных в программу поддерживают проверку формата вводимых данных. На рсунке 5.3 жирной линией подчеркнуто поле ввода, которое специально предназначено для числовой информации и не поддерживает ввод строковых констант.

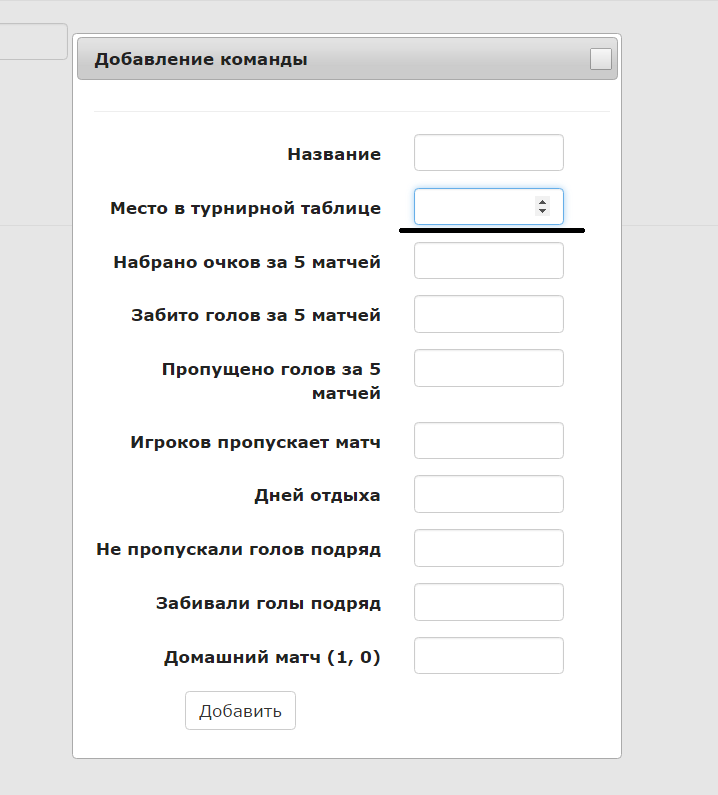


Рисунок 5.3 – Демонстрация валидации данных полем ввода

**5.3** Регрессионное тестирование

Регрессионное — проверка исправления вновь найденного дефекта, а такжепроверка, что исправленный ранее и верифицированный дефект не воспроизводится в системе снова. Сюда также можно включить проверку того, что не нарушилась работоспособность работающей ранее функциональности, если её код мог быть затронут при исправлении некоторых дефектов в другой функциональности.

Обычно используемые методы регрессионного тестирования включают повторные прогоны предыдущих тестов, а также проверки, не попали ли регрессионные ошибки в очередную версию в результате слияния кода. Поэтому считается хорошей практикой при исправлении ошибки создать тест на неё и регулярно прогонять его при последующих изменениях программы. Хотя регрессионное тестирование может быть выполнено и вручную, но чаще всего это делается с помощью специализированных программ, позволяющих выполнять все регрессионные тесты [автоматически](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). В некоторых проектах даже используются инструменты для автоматического прогона регрессионных тестов через заданный интервал времени. Обычно это выполняется после каждой удачной компиляции (в небольших проектах) либо каждую ночь или каждую неделю.

Данный подход используется в связке с модульным-тестированием, поскольку регрессионное тестирование может быть использовано не только для проверки корректности программы, часто оно также используется для оценки [качества](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%87%D0%B5%D1%81%D1%82%D0%B2%D0%BE_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) полученного результата. В данном случае проект представляет собой набор структурируемых тестов.