Integrating a Task Manager into an IDE

Bachelor Arbeit

der Philosophisch-naturwissenschftlichen Fakutlät
der Universität Bern

vorgelegt von
Michal Musial
Dezember, 2010

Leiter der Arbeit
Prof. Dr. Oscar Nierstrasz
Dr. David Röthlisberger

Institut für Informatik und angewandte Mathematik

Abstract

This project aims at integrating a task manager called *TaskManager* (TM) directly into the Visual Studio 2008 IDE. When developing a large software system, a developer typically spends a lot of time navigating through thousands of code artifacts to find the subset of information needed to complete the current task. The main goal is to associate code artifacts with the specification of programming features that are going to be implemented by developers, to increase the efficiency of their work. TM shows a ToDo list, but deeply related with the software system for which these tasks have to be performed (for example fixing a defect in that system or implementing a completely new feature). We can associate with every task software artifacts (classes, methods, etc.) we have to change, check, implement or remove from the system to achieve the task. For every task we also want to define tests that have to be added or adapted to fulfill the task. The task manager interacts with the rest of the IDE, for example to quickly navigate with the IDE to artifacts associated with the tasks, to add artifacts from within the IDE to the task. TM automatically changes the completion progress of a task when a developer works in the IDE on artifacts defined in the task. The empirical evaluation shows that TM meets the requirements of developers and it is seen as an useful tool which helps them during programming.

Acknowledgments

Thanks to my parents and to who believed in me during my studies.

I would like to express my sincere gratitude to my supervisor, Doctor David Röthlisberger for his support during the development of TaskManager and for his constructive comments regarding my work.

I would like to also thank my Professor Oscar Nierstrasz who inspired me to discover deeper topics related to developing software.

And a special thank for all persons in the M-S company who gave me advice about technical details during the development of TM.

Table of Contents

List of Figures

List of Tables

1 Introduction

The main goal of this project is the development of a plug-in deeply integrated into the IDE to increase the efficiency of developer's work.

1.1 Problem Statement

Commonly used IDEs like Visual Studio (VS) provide many possibilities to support the developer during programming. However, they do not establish a connection between the code and the specification of features which need to be implemented or corrected. The main problem touched by this project focuses on creating a plug-in, TaskManager (TM), which supports that kind of connection. The main goals are to increase the speed of programming by reducing time spent on navigating through code artifacts, making the source code easier to control and extend in the future. Furthermore, when using object-oriented programming languages the code relating to the same conceptually objects is distributed over many source artifacts such as classes and methods. Therefore it's hard to keep track of all the corresponding elements. TM helps the developer to avoid programming bugs or unwanted behavior in developed system. The following list shows the most important problems and activities:

- Identifying code artifacts like methods or classes with the goal to bind them together with the task.
- Giving the possibility to track the progress of tasks directly in IDE. Binding other programs like a bug tracking system to TM.
- Identifying various types of tasks such as fixing a bug or a new implementation and supporting the developer in programming respective of the type and context of the task.
- Supporting safe programming by generating tests and using them as important criteria influencing the completion of a task. How to design programs which encourage users to write tests that cover significant parts of the code?
- Building a team-oriented plug-in as an extension to VS. How should the communication between developers look like? How are they notified about any changes? How to implement a plug-in built in VS?
- Understanding communication problems that can occur in a team and how to solve them.

1.2 TaskManager – Binding Code Artifacts to Tasks in Visual Studio

Due to our experience we opted to use Visual Studio 2008 (VS) as an environment in which the implementation of TaskManager (TM) is done. Of the large number of programming languages supported by VS, we decided to use C#. TM is implemented as extension of that environment – a plug-in. The main problem of the project is related to

the task so at the beginning we can be clear that in the context of said project, the task is simply modeled as a set of code artifacts, which need to be created or modified to complete that task. To make it usable and readable tasks need to be defined directly in the IDE, where the task is completed. TM offers a possibility to define tasks with their name and to specify one of four task types. The type of the task determines steps which need to be taken to complete the task. During creation of the task the developer is also asked to select relevant for his task code artifacts. Once defined as task elements, TM will keep binding to tasks even if code artifacts change their name or are moved within the solution.

Besides code artifacts, a task involves also one test class containing the tests which should cover more or less the whole code related to the task. That test class is automatically generated when a task is created and extended by the developer when needed. According to Test Driven Development (TDD) satisfying tests contained in corresponding test classes decide about the completion of task. So the developer is in this context forced to write tests. It prevents the developed program from exhibiting unwanted behavior since the most important execution paths are covered by tests.

TM also offers the possibility to keep track of the progress of tasks. This is done by connecting each task with a web based bug tracking system, where the end-user and developers can communicate with each other. To support the multi-user interface, data needs to be stored in a database (DB) so in the end all users exchange information regarding tasks by sharing the same DB (Fig. 1.2.1). After each update, changes are automatically visible to the rest of team.

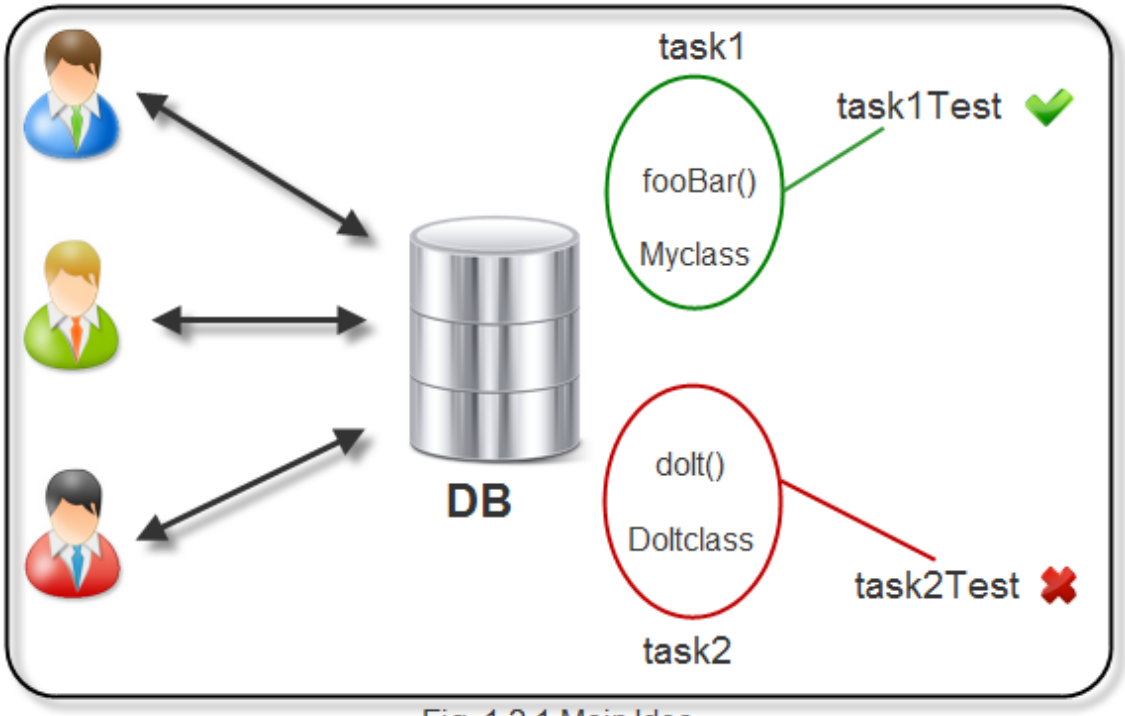

Fig. 1.2.1 Main Idea

1.3 Contributions

TM enriches VS by giving the possibility to keep code artifacts which are logically related together as a task. The process of extending VS requires many steps which need to be taken. In the following, the most important aspects are:

Reflecting object - widely used across the whole project with the goal to manage code elements within the code itself. Two main applications of reflection can be identified:

- Control over task elements – identifying and associating code artifacts with tasks, detecting changes on task elements.
- Code generation and execution of code to dynamically check whether the code satisfies the tests.

Binding code artifacts with tasks und users - using acquired knowledge about reflection to bind code artifacts with users using abstract concepts of the task. Binding is done to support communication within a team working on many problems.

Enrichment of VS – extending VS requires knowledge about the DTE Interface since that interface offers the possibility to add new functionality to VS. Enrichments include new menus and tool windows in the programming environment. TM improves navigation between related code artifacts within the VS.

Evaluation – TM was validated at work with the aid of a controlled experiment with eight experienced developers. They were asked to solve programming tasks using TM as a help tool. At the end we evaluated the usability of TM using measurements based on feedback given by developers.

1.4 Structure of the Thesis

We now give a short overview of the structure of the thesis.

Chapter 2 - explains what problems can be solved by our plug-in. At the beginning we compare TM with other existing solutions to find differences and to show that existing tools are not enough to solve our problem. We mention here also the advantages of using TM over base line solution offered by the VS. In this chapter there are also use cases for specific problems which show how the TM helps developers solve those problems and how to use our tool.

Chapter 3 - describes in detail how TM is designed and what it is exactly. Besides that we give information about the motivation for each design decision. In chapter 3 are all information about implemented features and a manual explaining how to use the features to correctly complete tasks and what is the idea behind them. At the end we write about the limitations of TM.

Chapter 4 - concentrates on the technical aspects of the implementation of TM. We also show how TM was created and describe the most important and interesting implementation challenges we need to take up during programming. We explain what

kinds of design patterns were used and how they help to solve particular problems. We also give a quick overview of how TM is integrated into the IDE.

Chapter 5 - presents how TM was evaluated and also the result of that evaluation. The presentation shows how and by whom TM was tested. This chapter gives also answers about developer's acceptance and what developers think about how well TM deals with problems presented within earlier chapters.

Chapter 6 – summarizes the whole project based on feedback from developers who used it. We present here conclusions about how TM works, in which situations it's applicable and for which tasks it is not recommended to use it. We also describe ideas for future work on TM. Chapter 6 contains a collection of interesting ideas proposed by the developers using TM.

2 State of the Art

2.1 Related Work

Several research tools provide functionality to help developers declare task relevant code artifacts. Most of the offered tools aim to decrease the amount of time spent on sifting through thousands of artifacts to find the subset needed to complete a software maintenance.

Mylar [1] is a tool which dynamically identifies relevant code artifacts by building degree-of-interest (DOI) trees. Mylar monitors programmer's activities and captures the relevance of code elements to his task in a DOI model. When a developer selects or edits a program element, Mylar increases the interest level of that element. Over time, if the element is not selected or edited, its interest value decays. Mylar can be used to narrow down the search field of the current task's relevant code artifacts. Since Mylar collects data dynamically it eventually becomes not precise enough to say which task belongs to the found code artifacts if programmer works on and switches between multiple tasks. In context of TM, the developer knows exactly which of the code artifacts are relevant. Also not accessed code artifacts are not going to be recognized as task elements. Even after enhancing Mylar in [2], the tool is not useful in the context of our project because according to the fact that Mylar decreases the level of interest over time, the tool cannot be used to get information about previous tasks.

NavTracks [3] proposes to exploit file-to-file relationships. The base concept of NavTracks is to understand the software space as a hypertext space with many different kinds of hypertextual relationships. Another assumption is that the information garnered from the navigation of source files can reflect a developer's mental model of the system. In the context of a task, NavTracks suggests files that may be of immediate interest to the developer. A list of files presented by NavTracks relies on the ability to recognize and not to recall relevant code artifacts. Unlike TM, NavTracks does not consider relationships within the source file. This means that in case of files with a big number of code artifacts TM offers more precise information about task's elements. Having a map showing relationships between files, it can be hard to correctly identify tasks themselves. The suggestion list offered by NavTracks is helpful if the developer can recognize the code files from a previous session, otherwise he needs to check each suggested file, which only narrows down the search field.

Team Tracks [4] was developed by Microsoft and is a tool providing visualizations of related code artifacts based on shared navigation. It is the first solution designed especially for teams. Team Tracks helps team members to get a better overview of the unfamiliar source code. The most important assumption make by Team Tracks is: the more often two parts of the code are visited in succession, the more related they are. The solution proposed by Microsoft is very similar to NavTracks. Unlike NavTracks, Team Tracks shows also a suggestion list of related code artifacts within the source file, so a developer is able to see relations between objects and methods. Nevertheless, suggested relations can relate to completely different tasks and it can be hard to distinguish between the tasks if as splitting criteria we have only relations between code artifacts.

Belotti [5] shows concepts related to the development of a task list manager system that could help users manage and execute their daily to-dos. To-dos are not defined directly and can be different from programming tasks but reading Belotti's document helps to understand what is important to represent a task. Important issues proposed by Belotti we considered when developing TM are: When designing task management system identify involved subjects, give the possibility to express needed effort, define level of details describing a task, give the possibility to reuse existing features, support many possibilities to give an input, define factors deciding about when a task is completed.

FEAT [7][8], was developed by Robillard and Murphy. FEAT provides a mechanism for explicit documentation of tasks distributed in the program. FEAT uses a concern graph. The concern graph can be used for navigating to related code artifacts in a concern. A concern can be understood as a task. To create graphs, developer intervention is required to define a new concern. FEAT shares the same goal as TM to improve navigation. Unlike TM, FEAT allows us to define only one type of task and provides no information about the developers involved in the evolution of the task. FEAT assumes that a developer works on only one task at a time. Detection of related code artifacts decreases in precision if the developer switches between tasks.

2.2 Improvement of the Visual Studio's Internal Solution

VS provides by itself some kind of ToDo list called *Task List* but it has limited functionality. There are two main parts which belong to *Task List*: *User Tasks* and *Comments*. Both parts are only marginally related to each other and designed to solve different problems.

All that can be done with *User Task* is writing a description, toggling a binary status, using a checkbox to complete or not complete and defining the priority of a task by choosing one of three levels (high, normal, low). In the solution proposed by Microsoft, the creator of the task decides when the task is done. Tasks are defined locally for a single developer. The Motivation behind *User Tasks* is to enter notes on work to be done. Fig. 2.2.1 shows an example of *User Tasks*.

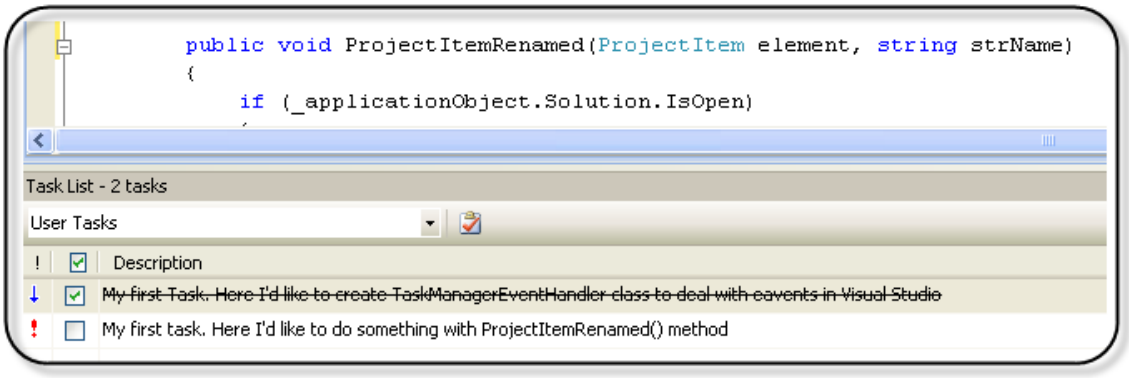

Fig. 2.2.1 User Tasks in VS

As we see in Fig. 2.2.1 there is not enough space to write a complex description and it's also hard to reuse information regarding previous tasks. The task itself contains very little relevant information.

Comments can be used to mark lines in a code file (but not a code artifact itself) to be modified or implemented. To mark the lines of code developer has to first to specify the token. A token can be any single word. An example of using *Comments* in VS is shown in Fig. 2.2.2 where the token "TODO" was used:

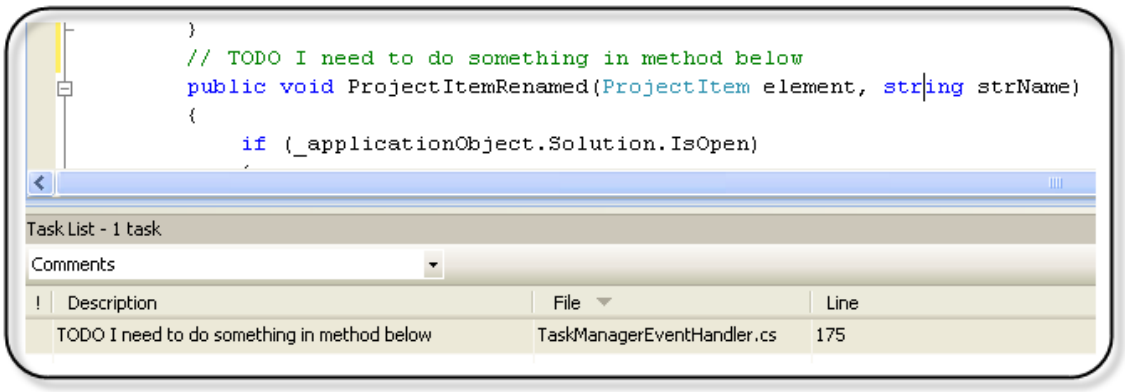

Fig. 2.2.2 Comments in VS

To create *Comments* the developer needs to first to specify a token and then write a comment next to the code that has to be modified the comment needs to start with that token otherwise it will not be recognized by the *Comments List*. Each token can have a predefined priority. Tokens and their priorities are defined once again locally per developer and not globally.

The first problem of the built-in solution is that *User Tasks* are only visible for a single developer. *Comments* instead are visible for the entire team only if each developer defines same tokens in his instance of IDE. The idea behind *Task List* is local usage and it is not recommended by design to use *Task List* globally. Another problem is that there is no possibility to associate *Comments* with *User Tasks* this means that they are separate features since there is no relation between them. *Comments* don't represent code artifacts, they are just in their neighborhood. So there is no possibility to detect if code artifacts they are referring to change or not. If they change we should perform some action which will change the status of the associated task and make history entry to document the progress of the task. If there are no global definitions of *User Tasks* or *Comments*, using the *Task List* brings only advantages for single users but there are no benefits for the whole team since there is no possibility to exchange information between the team members. The feedback about the progress of the task is also modeled very simply and developers cannot get any precise information about state of developed task. Developers need to check if the task is completed or not and manually change the state of the task. There are no tests generated to check correctness of the implementation they need to be created separately.

These observations lead to the conclusion that the existing solution cannot be used for our goal, because the concept of a task is completely different. The solution proposed by Microsoft interprets a task as a small problem which can be described in few words, which is probably only for current usage and needs to be solved in the nearest future. At the end the task will be removed from the *Task List*. In reference to the need of binding the code artifacts directly to the tasks a completely new design is re-

quired, where tasks are defined globally for all users and contain real code artifacts to solve more complex problems.

2.3 Expected Benefits

Programming Faster

The main goal of our project is to build an application which helps the developer to increase the speed of developing new systems but also prevents programming bugs and helps developers to find all needed information so that they can better concentrate on programming. It's done by binding specification of features which need to be implemented or corrected with code elements. Decreasing the time needed for navigation through lines of code reduces the time needed to implement features. Another goal is to keep logically independent parts of the code in one place called a task, which in the end gives developers a better overview of the system.

One of the most important advantages is that programming becomes safer. By design of TM we preserve the developer to implement tasks according to TDD. That means that in an ideal case each line of code is automatically covered by tests, which makes detection and correction of unwanted program behavior much easier. In case of unwanted behavior a test simply fails and the developer can reproduce the error by executing the corresponding test cases and knows exactly which preconditions are broken and which code elements are involved in this bug which makes correcting easier.

Using TM information about the task and associated code artifacts can be found and reused quickly. TM can be used not only for the current task but also to view the history about previous activities. A simple use case is when a developer is asked to implement something similar to that what was done in the past he receives a link to the corresponding task and can navigate to code artifacts and so get to know about implementation details.

Statistics about Efficiency of Programming

Another advantage of using TM is giving statistics about tasks, which can be interesting if there is a similar problem to solve in the future. Having information about previous needed effort and design problems, the team can better estimate the required effort for correcting problems, reuse code or ideas from the past. Statistics can also be used to give an overview of the efficiency of developers involved in the completion of a task and to check whether specified deadlines are held.

Understanding Communication within the Team

The result of our work team-oriented software should give an idea about the most important communication problems which can occur in a team and how to solve them. Designing a system with a clearly structured information flow gives the possibility for the

developers to see how tasks are distributed over team members. This improves information exchange within the team.

3 TaskManager

In Chapter 1 we listed several problems we aim to solve. According to that we implemented TM as an extension of VS 2008 – a plug-in. Using multiple views, TM allows developers to bind code artifacts and developers to tasks towards the goal of making programming easier and faster. After starting VS our tool can be found in the tools menu bar on the top. The location of TM is shown in Fig. 3.1.1.

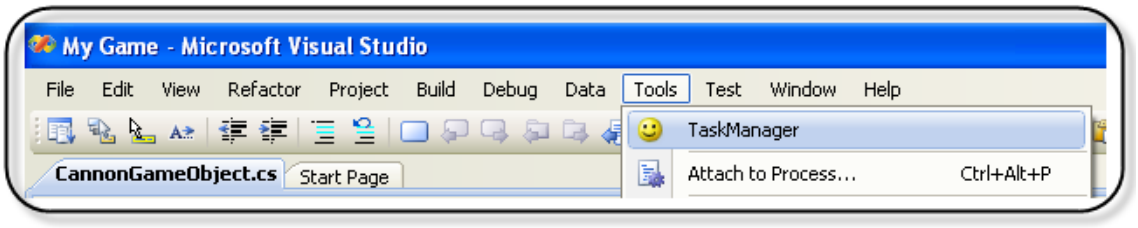

Fig. 3.1.1 TaskManager Location

3.1 Main Idea

As described in the introduction the main goal of this project was to develop an application that allows developers to assign code artifacts, tests and developers to the concept of a task. At the beginning we explain precise definitions of each actor and the main role it plays in TM.

Code artifact

A code artifact is a piece of source code written by a programmer. VS offers the following hierarchy of code artifacts from top to bottom:

- Solution - entire program.
  - Project – one component of solution for example logic or tests.
    - Namespace – a package with a number of classes.
      - Class.
        - Method.

In the context of TM the following source code artifacts are seen as code artifacts:

- Public method
- Public class
- Public abstract class
- Interface
- Packages

Only public members are shown in the list of code artifacts because showing all members in the list of code artifacts makes it not readable enough. Another argument is that public members define the final functionality offered by the interface. Private members can be associated with a task if parent public member is chosen as a task element, for example a public class containing private methods which have to be implemented.

Test Case

A test case is represented by one of the methods contained in a test class. Each of the methods defines its own test case. In our design, a task is associated with only one test class which should contain all test cases related to that task. Test class with template test cases for each code artifact is automatically generated by TM.

Developer

The developer is the person who uses TM. His goal can be either the development of new features or the correction of existing ones.

Task

A task binds together all elements listed above. There are four kinds of tasks we propose in our solution:

- New Implementation
- Bug Fix
- Test
- Refactoring

Therefore we can describe the main goal of the task and by choosing its type decide about the future progress of the task.

A task has many other properties which can be specified at any point in time. Other functionalities of a task are:

- documentation of the progress
- simple statistics about the evolution of the task

3.2 Use Case

TM offers four kinds of tasks that can be defined. By describing a simple scenario we give an impression about the tool, when it is most applicable and what kind of problems can be solved with it.

Star Battles - Problem Statement

John and Mark develop a game – Star Battles. In the game, players can fly with the space ship and destroy hostile space ships. The main part of the game is already implemented but there are two things left to do. Developers would like to extend the game with a new functionality – cannons that destroy enemies should support more fire modes than just one. Moreover, users of the game reported in a bug tracking system that there is only one type of enemy in the game, however Mark is sure that he programmed the game to support two types. They decide to use TM to help them solve those two tasks. John decides to implement new functionality concerning the cannons and Mark wants to fix his code.

Step 1: Connection with DB

To begin to work with TM, first of all the database connection should be established. This can be done by specifying server name, user name and password if needed. Since TM generates the required tables, the tool is ready to work. Details in Fig. 3.2.1.

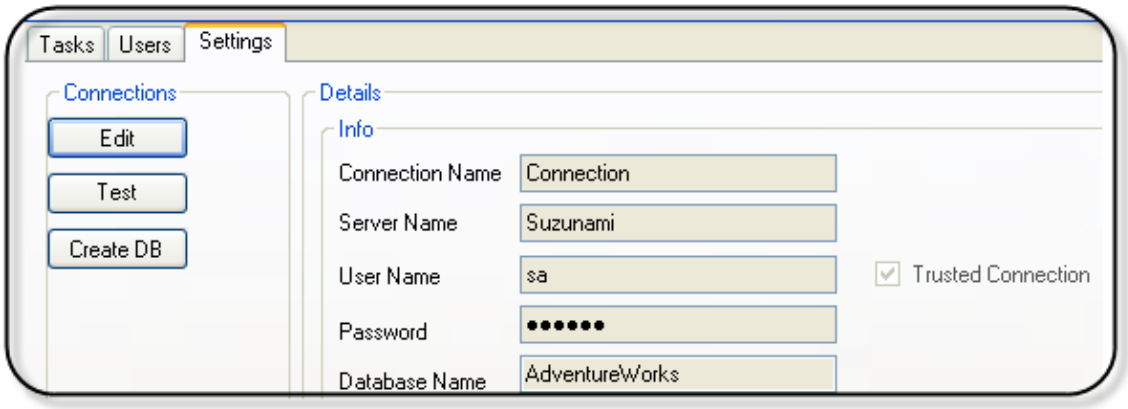

Fig. 3.2.1 Connection with DB

Step 2: Define User

Since John and Mark use TM, they need to create themselves as developers. They are asked to give their names and other necessary data. Details in Fig. 3.2.2.

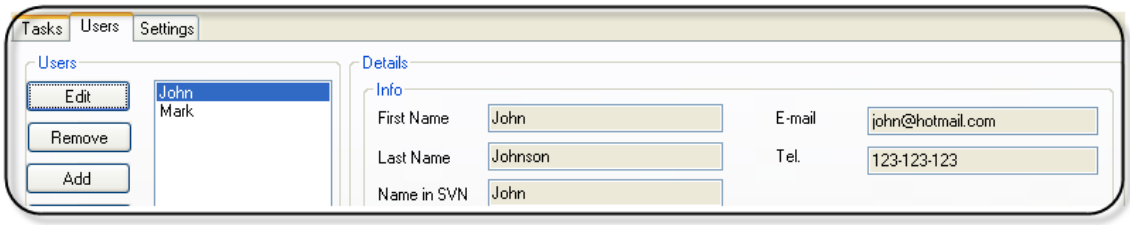

Fig. 3.2.2 Define User

Step 3: Define a Task

In the *Task* section they can create tasks as described in Chapter 3.2. John defines his task of type *New Implementation* since he wants to add new functionality. He calls the task "Laser cannon". Mark creates another task of type *Bug Fix* called "Missing enemies", he also assigns his bug to a bug tracking system and uses TM to write a message to the bug reporter to inform him that the problem is going to be solved in the near future. Details in Fig. 3.2.3.

Step 4: Assign Existing Code Artifacts

Mark assigns the EnemyGameObject class with all methods to his task since this is the only class where enemies are created. John assigns CannonGameObject class with its Fire() method to his task. Details in Fig. 3.2.3.

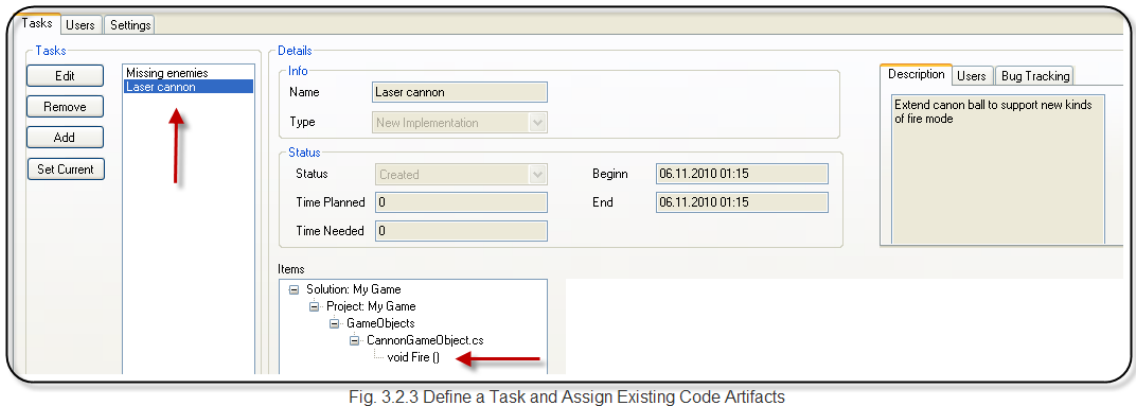

Fig. 3.2.3 Define a Task and Assign Existing Code Artifacts

Step 5: Write Tests

Mark writes his tests in the generated test class. In his test case he expects that after starting the game there should be two different enemies available. John writes two tests where he expects that when ten enemies are killed, a laser mode is on. In the laser mode hundred cannonballs are available instead of five as before. This should simulate the effect of the laser. Details in Fig. 3.2.4.

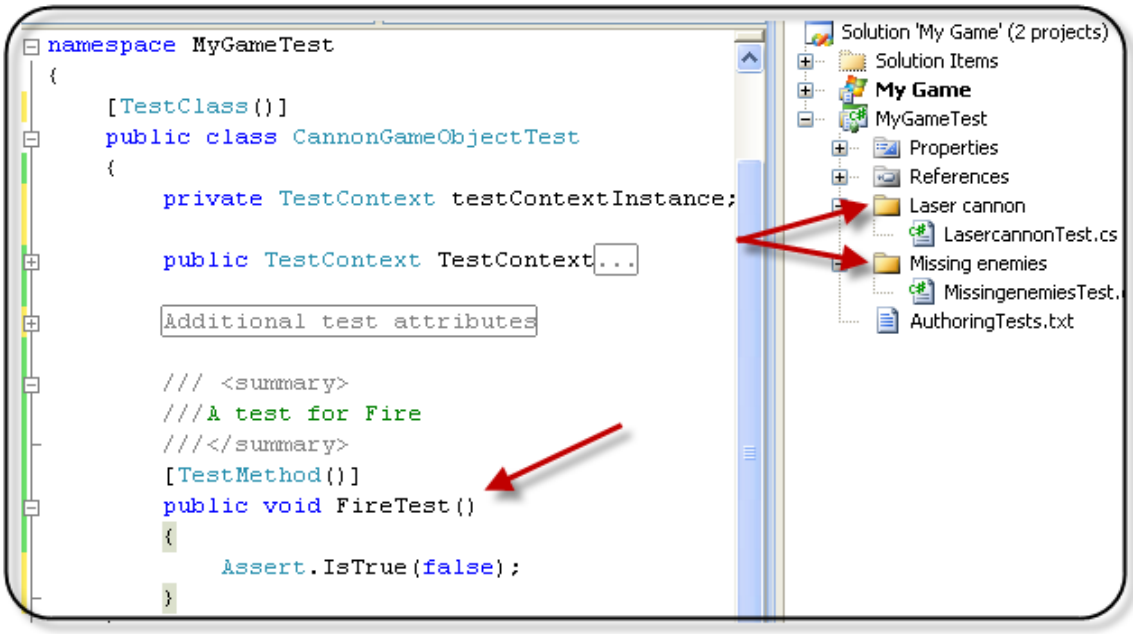

Fig.3.2.4 Write Tests

Step 6: Write Code

John modifies the Fire() method to activate the laser mode as soon as five enemies are dead. In the next step he creates a new class LaserGameObject which shoots hundred cannonballs at once. He assigns that new class to his task directly in the code view using the right click menu. Mark found the bug in his code – he called the method CreateEnemy**1**() twice, while method CreateEnemy**2**() was never called. He corrects the code. To avoid misunderstandings in the future, he decides to rename the methods and give them some more meaningful names. To do that he uses the rename tool offered by VS which updates automatically all references in TM. Details in Fig. 3.2.5.

```
if (enemiesKilledCount == 5 && !specialWeaponDroped)
{
    specialWeapon.position = new Vector2((float)random.NextDouble() * ViewportRect.Width, 0);
    specialWeapon.velocity.Y = 5.0f;
    specialWeaponMode = true;
}
```

Fig. 3.2.5 Write Code

Step 7: Update TM

John starts TM and sees that his task receives the status ‘completed’ since all tests are satisfied. Mark sees that his task has also status ‘completed’, so he informs the bug reporter about changes using the bug tracking system connection offered by TM. John and Mark check if any open task exists in TM and realize that everything is already done. Details in Fig. 3.2.6.

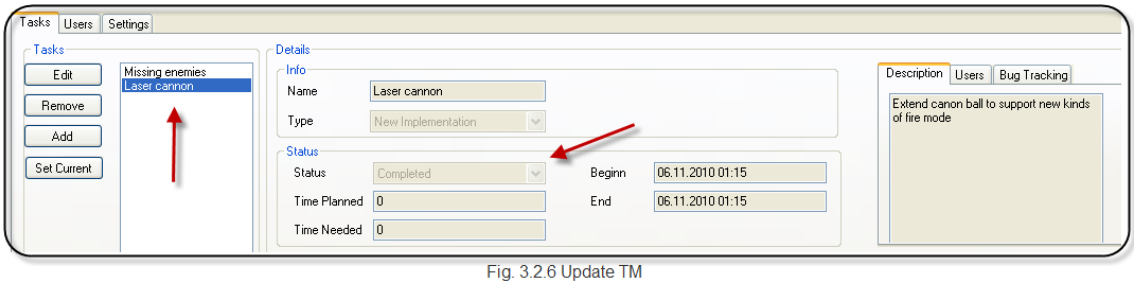

Fig. 3.2.6 Update TM

According to incremental design when solving some more complex tasks steps 4,5 and 6 can be repeated many times to complete a task.

3.3 Implemented Features

Creating Tasks

A task can be created to represent a problem which needs to be solved by one or more developers using the IDE. A task gives the possibility to express that problem by writing a description and binding that problem to a bug tracking system. The bug tracking system is used as one of the options to document the progress of a task. Any arbitrary web-oriented bug tracking system can be chosen and managed within TM since it offers a built-in web browser. A task also offers the possibility to associate code artifacts and developers with it. Simple statistics are represented by properties such as time planned and time needed, begin or end date. The overview of the task tab where developers can create their tasks is shown in Fig. 3.3.1.

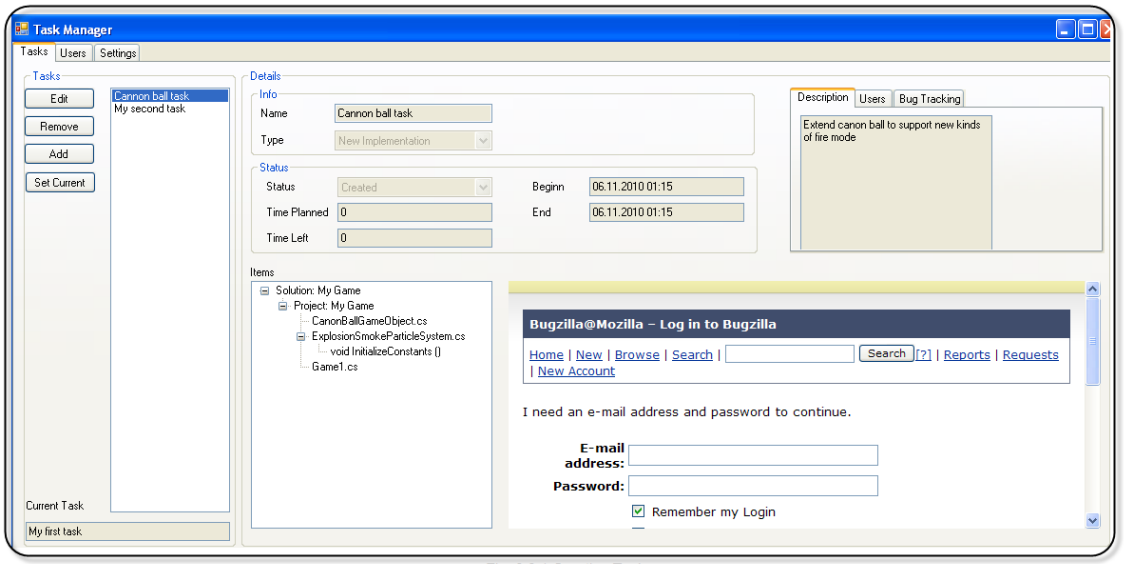

Fig. 3.3.1 Creating Task

When creating a task, the developer is asked to specify one of four task types each of which is applicable to different situation:

- **New Implementation** – implementing completely new functionality or extending existing one, but still in the context of creating new functionality.
- **Bug Fix** – some defect in code that leads up to unwanted behavior of program and have to be corrected. Note that functionality already exists so no new feature is going to be added.
- **Test** – some existing functionality has to be tested to verify that it is working properly by writing suitable tests. No implementation needed here since the code already exists. Note that the code was created probably without using TM since TDD would guarantee the existence of tests.
- **Refactoring** – code needs to be modified without changing its functional behavior in order to increase readability, decrease complexity of the code, to improve maintainability or to make program code more flexible for extensions.

The type of a task determines whether a test class will be generated by TM or not. Generation of tests has an influence on TM decision: "is the task already completed or not?". For the first three types of tasks a test class is going to be generated. In case of Refactoring, tests are not generated since the meaning of that type is that some code has to be reorganized and no new functionality is going to be implemented. Of course all code should be covered with tests but in the context of refactoring, we assume that those tests already exist and should be eventually adjusted to satisfy refactored code.

Tasks are defined globally per solution in the *Tasks* section of the TM. It means that a task is visible for all developers in a team, so everyone can manage them. But it also means that they are visible only within the solution they were created in.

When creating tasks automatically, a test class and if needed a separate test project, are created. A test class with a number of test cases decides about the completion of the task. In the first version, TM itself could make that decision. However after the evaluation, we decided that developer should also have that possibility (details in chapter 5). A list of all tasks available in the solution is visible on the left in the *Task* section in Figure 3.3.1.

Current Task

The idea of the current task is to specify the task that is already chosen by the developer to work on it. At any point in time there can be only one current task. This current task is represented as a window which shows information about the assigned code artifacts and developers. Developers can place the current task window anywhere in the IDE. The current task window shows only the necessary information about the task for details TM has to be started. The current task window cannot be too complex since it's placed in the code view where many other tool-windows share the same space. We decided to allow only one current task. The reason for that was that a developer mostly works on one problem at any point in time so there is no need to switch between tasks at that point.

Associating Code Artifacts

The main feature of TM is to associate code artifacts with tasks. Binding is done by saving a reference to specific source artifact in a DB. There are three main possibilities to create a binding between a task and code artifacts:

- Binding in TM – developer edits an existing task or creates a new one in the *Tasks* section of TM. Then he is able to choose from all solution elements represents as solution tree code artifacts that he wants to assign. Note that in a solution tree only public members are listed because showing all elements makes the tree less readable. Binding code artifacts in TM is shown in Fig. 3.3.2.

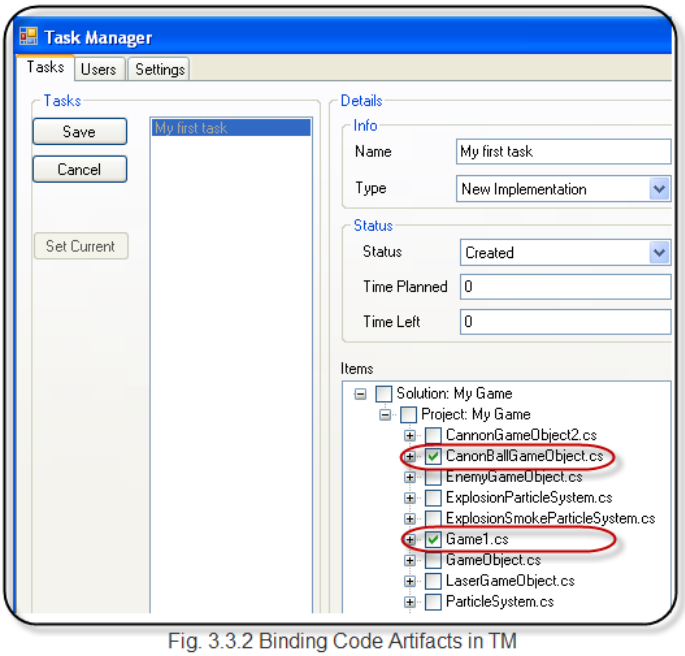

Fig. 3.3.2 Binding Code Artifacts in TM

- Binding in code view using the right click menu – the developer can choose one or more code artifacts by selecting them directly in the code view and then assigning them to a task using the right click menu. From the list of tasks he chooses the specified task. Binding code artifacts using the right click menu is shown in Fig. 3.3.3.

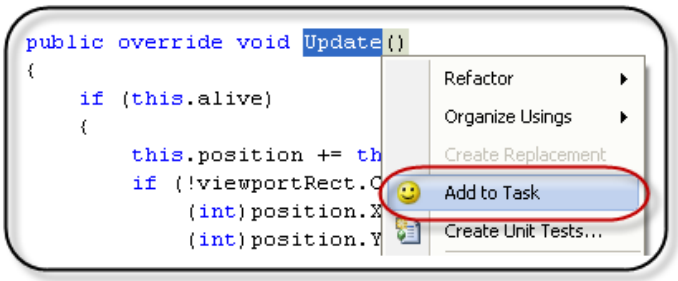

Fig. 3.3.3 Binding Code Artifacts using the right click menu

- Binding in code view using a shortcut - the developer can choose one or more code artifacts by selecting them directly in the code view and assign them by using shortcut Ctrl+Shift+Alt+A. A shortcut can be redefined in Tools->Option->Keyboard. When redefining shortcuts in VS, the developer is asked for a command namespace in case of TM namespace is called "TaskManager".

Control over Code Artifacts

TM has to control code artifacts at runtime. Since code artifacts can change the most important issue is to keep a task's reference to the code artifacts in a consistent state. Each time when source code referring to the code artifact changes, TM takes effort to update the DB to keep it in a consistent state. The main advantage of doing that is

that the task's reference to the code artifact is kept even if source code changes during the progress of the task. TM guarantees consistent state after executing following actions:

- **Renaming** of code artifacts using the rename command – All child references are also updated. Note if the developer renames some task elements to reuse them in another context than defined by the task, he is himself responsible to remove the binding from task.
- **Shifting** code artifact within file – when the class is reorganized and source code changes its position within the file.
- **Delete** – when a code artifact is going to be deleted its reference in any task is also deleted. If the associated task element cannot be found within a solution, the DB needs to be updated as well and the orphaned element is going to be deleted. The reason for that behavior is that a missing element was changed outside the solution or not in a "desired way". The context of using the code artifact is different from that defined by the task or the developer broke the precondition – for instance, the rename command was not used. Taking no action at that point could lead to inconsistent state in the.

Creating Users

Tasks can be created and managed by the developers. Developers are defined globally and are visible for all solutions, since probably the same users implement many different solutions. Creation and Editing developers can be defined in section *Users* which contains two boxes, *Info* and *Tasks* . All necessary information about a developer are grouped in the *Info* box. *Tasks* box gives an overview of a developers tasks. A list of all available developers and attributes describing a developer are visible in the *User* section as in Fig. 3.3.4.

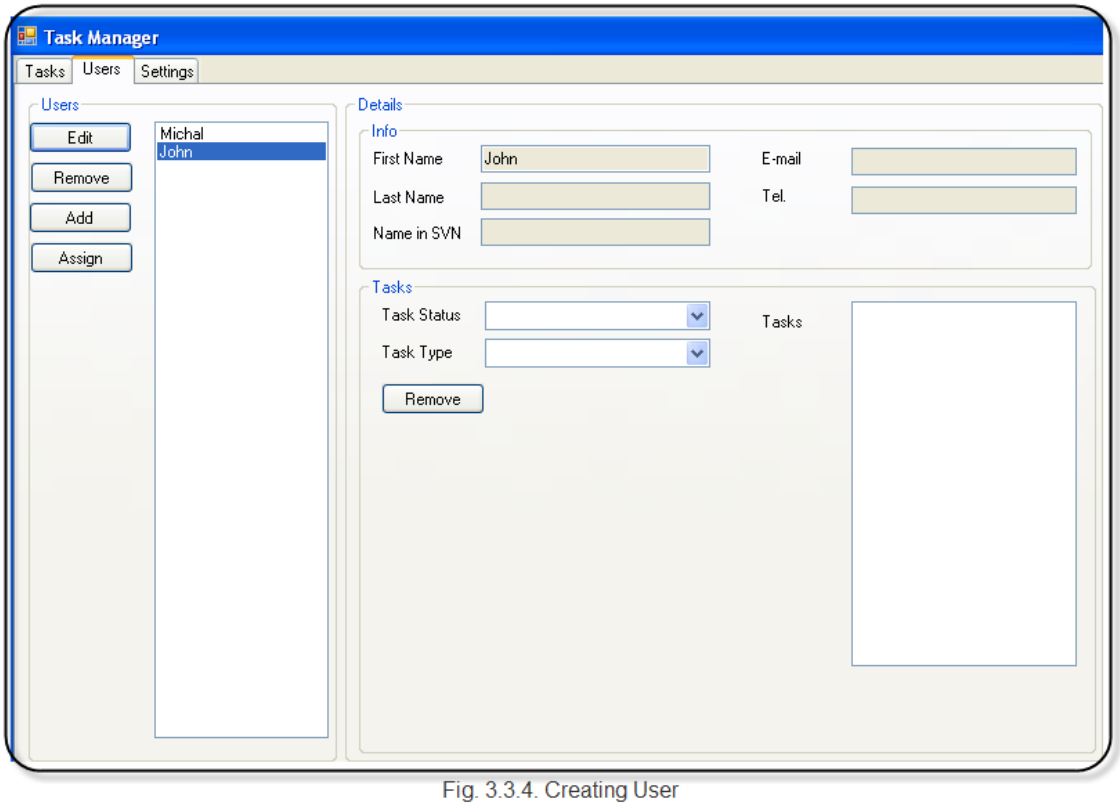

Fig. 3.3.4. Creating User

In our implementation there is no specification what role the user plays in the completion of the task. All users are defined in the same way and everybody can create them. Created users can be assigned to a task.

Assigning Developers to a Task

The next step is to connect tasks with developers to define who is responsible for a task. This binding reference is going to be saved in the DB. Since there is no user hierarchy or permissions, everyone can do this assignment.

The solution proposed by us offers two ways to assign developers to a task. The relation between users and tasks is n:m, so we decided to split it into two times 1:n. From the developer‘s point of view we are able to choose tasks they are interested in. Assigning developers in *User* section in TM is shown in Fig. 3.3.5. On the other hand, we can operate from the task point of view where we can specify which developers should belong to a task. Assigning developers in *Task* section in TM is shown in Fig. 3.3.6. Both operations give the same results. When the binding between tasks and developers is done, developers become visible under *Users* in *Task* section. The same holds for a task in its *User* section. Besides that, the *User* section offers the possibility to filter the view of user’s tasks using as criteria task type and task status.

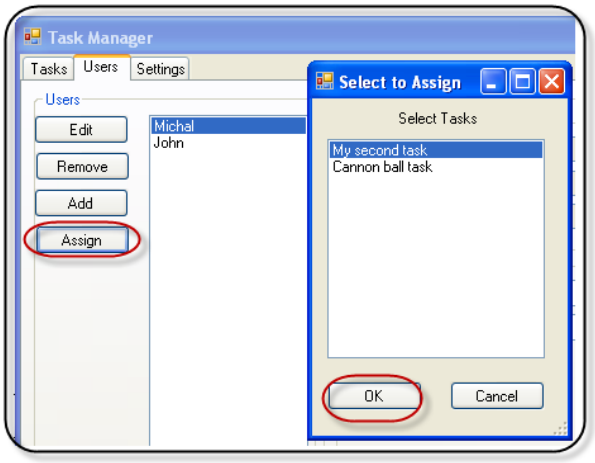

Fig. 3.3.5 Assigning User - Users Section

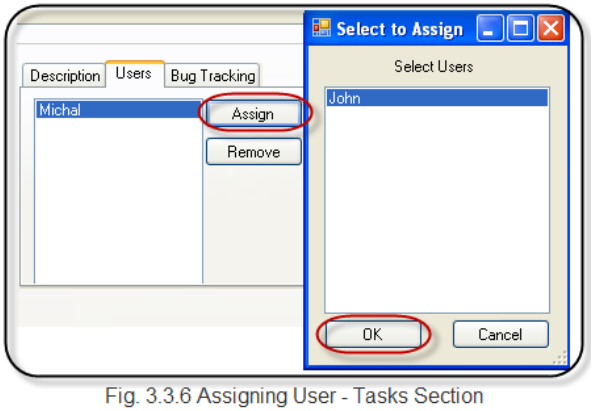

Fig. 3.3.6 Assigning User - Tasks Section

Test Generation

When creating a task, a test class and test method templates corresponding to each code artifact are automatically generated. They are a very important part of TM because by default they decide if a task is accomplished or not. This supports the idea of TDD where tests should be written before implementation is done. By requiring developers of TM to use TDD we achieve two important goals of the project: make programming easier and safer.

The first benefit of using TDD is that most of the code is covered by tests, thus programming with TM becomes safer. Another advantage is that developers have to continually repeat very short development steps to achieve their task. In the first step developer needs to write failing tests corresponding to use case that defines a desired

behavior of a new feature or feature extension. TM helps in that step by generating a test class with its methods. If needed the developer can extend the class with additional test methods but they need to be defined in the generated test class. In a second step, the developer is asked to write code to satisfy the tests. The third step is to run tests and check if they succeed if yes the task is done or the developer is ready for the next design iteration. In the latter case the developer goes back to the first step and defines new tests. If some of the tests fail the code has to be fixed. As the evaluation shows (Chapter 5) dividing programming into small steps like described above encourages simple design and increases confidence in code which in the end makes programming easier and faster.

Documentation of Tasks

The idea behind documentation and its main benefit is to improve communication in the entire team and to give developers the possibility for reusing tasks in the future if similar requirements occur. Each person in the team is able to find out which tasks exist and what their progress is. The possibility to document a task and the contribution of TM in case of reusing information are given below:

- **Description** – the developer can write a specification and exact description of the task.
- **Bug tracking system** – since TM can be connected to any web-oriented bug tracking system, a developer and an end-user of the developed software can communicate using the bug tracking system. All messages sent between the developer and the end will be saved and available for the future.
- **Code artifacts list** – all information about code artifacts associated with a task is stored in the DB as long as the task exists in TM. Code artifacts themselves provide all needed information about implementation details. If a developer is interested in technical details he just needs to navigate to relevant code artifacts. TM gives corresponding links. Therefore tasks should never be deleted, otherwise information cannot be reused in the future.
- **Tests passed** – simple statistics about how many tests are already satisfied compared to the total number of tests This information is used to predict the progress of the task. Note that information about passed tests does not say exactly how much of the task is already done because the complexity of sub activities covered by tests which need to be done to complete a task can vary strongly.
- **Task properties** – when creating a task, the developer is asked to give some data concerning estimated effort. At the completion of a task, the developer can write what was the real effort. Having that data contributes to the prediction quality in the area of estimating needed effort in the future and helps the team to identify developers who meet their deadlines.

To learn from previous tasks in a project, a developer just needs to know who implemented a particular feature or what the task was. Once he knows that, he opens TM to find the corresponding developer and navigates to his tasks or looks at the task directly.

3.4 Limitations

As described in previous chapters, TM can be a useful tool when developing software. But there are also situations when TM cannot work properly – it loses a task's references to associated code artifacts. VS keeps each solution object-oriented, this means that classes and their methods are treated as objects. This makes detection of changes easy (as long as changes were done in the desired way). When an object is changed VS sends an event, which can be handled with the associated `EventHandler` – in case of TM used widely to control code artifacts. "In the desired way" means here using VS tools to modify a solution element, otherwise the event is not being sent and cannot be caught which leads in the end to an inconsistency between available code artifacts in the solution and data saved in the DB. The references are not going to be updated so code artifacts cannot be identified later. An idea to improve that limitation is described more precisely in Chapter 6.

4 Evaluation

4.1 Motivation

We decided to conduct a controlled experiment with eight professional developers who have experience in using C# in VS. The experiment's purpose was to validate our claims that TM:

- Decreases time spent on solving tasks.
- Decreases time needed to understand the code.
- Improves information exchange within the team.

Our experiment was made early enough to improve TM based on experiment results and feedback given by developers. To get reliable results and to see differences between using VS without our tool and with TM, we tested our hypotheses on two groups of subjects: an experimental group and a control group.

4.2 Experiment

Subjects

We asked eight C# developers working in the same company to solve our experimental tasks. To solve daily tasks the company uses VS 2008 as an IDE and as a programming language C#. Subjects were asked to fill out a questionnaire asking about their experience in using C# and VS and overall consideration about features offered by TM. Subjects were divided into two groups with similar average experience in using C# and VS:

Control group – a group of four experienced C# developers, who were asked to solve tasks without using TM. Details regarding the control group are presented in Table 5.2.1.

Subject	Years of experience in C#	Years of experience in VS	Age	Years in company
Subject 1C	6	8	35	8
Subject 2C	7	10	44	10
Subject 3C	5	7	31	6
Subject 4C	7	10	37	8
Mean	6.25	8.75	36.75	8

Table 5.2.1. Control Group

Experimental group – a group of four experienced C# developers were asked to solve tasks using TM. The experimental group was introduced to all features offered by TM and instructed how to use them. Before the experiment started, we allowed subjects to play with TM to get know our tool. Details regarding the experimental group are presented in Table 5.2.2.

Subject	Years of experience in C#	Years of experience in VS	Age	Years in company
Subject 1T	4	6	29	4
Subject 2T	5	8	33	9
Subject 3T	7	9	35	8
Subject 4T	5	9	42	9
Mean	5.25	8	34.75	7.5

Table 5.2.2. Experimental Group

Tasks

We prepared a special test application – a game called “Space Battles”. The application contained 20 classes and 167 methods. A more detailed description of the game can be found in Chapter 3.2. The code of the test application was unknown to all subjects. At the beginning each subject could play the game to see how it works. The experiment tasks were divided in two groups: programming tasks and a questionnaire. We modeled our programming tasks so that they involved following activities, which subject had to perform to solve the task: writing the code (WC), writing the tests (WT), learning about the unfamiliar code (LUC), navigating code artifacts (NCA) and for the experimental group only, assigning code artifacts to the task (ACA). Subjects were asked to solve the following tasks, important activities are mentioned in parentheses:

New Implementation – In the first level of the game there should be a new kind of enemy who needs to be shot five times from the cannon to be destroyed. For the new functionality we write two tests that verify the desired behavior. (WC), (WT), (LUC), (ACA), (NCA)

Bug Fix – After killing ten enemies, a laser falls from the sky to be caught by the player. The laser is not activated. Correct that behavior and write tests which verify that the laser is activated. (WC), (WT), (LUC), (NCA)

Test – After collecting 100 points, the player moves up to the second level. The behavior of the game works correctly. Verify it by writing tests. (WT), (LUC), (NCA) , (ACA)

Refactoring – There is no common interface for game objects like player or enemy, those are able to move. Refactor all movable game objects by introducing a suitable super class. (WC), (LUC), (ACA), (NCA)

Information exchange – Discover and describe how other group members solved their tasks. To solve this exercise read the code and (if the subject is in the experimental group) use TM. (LUC), (NCA)

Fill out questionnaire about TM (only experimental group) – Subjects could write the note and their commenter. The questionnaire is shown in Table 5.2.3.

1.	TM is easy to use (10 = Agree; 1 = Disagree)
2.	TM is easy to learn (10 = Agree; 1 = Disagree)
3.	Binding code artifacts to tasks directly in IDE reduces the effort needed to solve a task (10 = Agree; 1 = Disagree)
4.	The related code artifacts list is useful for rapid navigation through the code (10 = Agree; 1 = Disagree)
5.	TM helps developer to better and faster understand unfamiliar code (10 = Agree; 1 = Disagree)
6.	TM helps to keep task specific elements together (10 = Agree; 1 = Disagree)
7.	Generating tests helps developer to solve task (10 = Agree; 1 = Disagree)
8.	Task completion should depend on satisfied tests (10 = Agree; 1 = Disagree)
9.	TM helps developer to track the progress of the tasks (10 = Agree; 1 = Disagree)
10.	TM helps developer to learn from previously solved tasks (10 = Agree; 1 = Disagree)
11.	How satisfied were you with the TM? (10 = High; 1= Low)

Table 5.2.3 Questionnaire about TM

The control group received the specification of the tasks on paper and the experimental group in form of task definitions in TM. We supervised the subjects during the entire experiment, and for each programming task we recorded the time spent on each activity.

Results

First we evaluated the time results of each activity. From those results we build the final conclusion how TM influences the mean time spent on a task and mean time spent on each activity. Tables 5.2.4 and 5.2.5 presents an overview of our results.

Experimental group	WC	WT	LUC	ACA	NCA	Total
New Implementation	6.43	2.57	6.32	1.11	2.24	18.67
Bug Fix	3.67	2.96	5.57	0	2.13	14.33
Test	0	6.48	7.17	1.94	1.96	17.55
Refactoring	4.53	0	3.41	1.4	1.04	10.38
Information exchange	0	0	8.49	0	2.2	10.69
Total	14.63	12.01	30.96	4.45	9.57	71.62

Table 5.2.4 Time Results for Target Group per Activity

Control group	WC	WT	LUC	NCA	Total
New Implementation	6.13	4.53	6.15	4.51	21.32
Bug Fix	3.94	3.79	6.17	3.37	17.27
Test	0	8.21	8.12	4.21	20.54
Refactoring	4.15	0	4.5	2.5	11.15
Information exchange	0	0	8.62	3.5	12.12
Total	14.22	16.53	33.56	18.09	82.4

Table 5.2.5 Time Results for Control Group per Activity

From the results presented in Tables 5.2.4 and 5.2.5 we can conclude that the experimental group solved tasks 13.08% faster than the control group. The reason for the speedup was 47.10% reduction of time needed to navigate code artifacts. Thanks to the TM the developer knows exactly which code artifacts are relevant. The second reason for the speedup was a 27.34% reduction of time needed to write tests, since TM supports developer in this activity by generating test templates. The last reason for the speedup was a 7.75% reduction of time needed to understand unfamiliar code. We believe that this reduction is achieved thanks to NCA time reduction. Subjects could better concentrate on reading the unfamiliar code than on navigating code artifacts and because of that the subject's short time memory was more optimally used. We did not notice any relevant influence of TM on time spent on WC. The summary about time result differences is shown in Table 5.2.6.

Total Time Spent Difference	-13 %
WC Time Difference	3 %
WT Time Difference	-27 %
LUC Time Difference	-8 %
NCA Time Difference	-47 %

Table 5.2.6 Time Results Differences

The results given by the questionnaire are also satisfying. Our tool got an average grade of 7.84, where 10 was the maximum possible score. According to the questions asked in the questionnaire, we believe that TM meets developers requirements and is considered by them as a useful tool when solving programming tasks. Subjects were convinced that TM increases the navigation speed and helps them to keep independent parts of the code together. The comments to the open questions gave us a feedback about our tool. This feedback motivated us to give the developer also a possibility to decide about the completion of the task. The reason for this is that in industry it is often hard to test all software behavior and if the entire feature cannot be tested, for example because of deadline reasons, setting status to complete based only on a few tests can be dangerous. Subjects suggest removing the specification of some properties of a task such as create date, end date or the possibility to filter the task list by using those properties as filter criteria. Comments given by subjects inspired us to define new extension possibilities The most interesting ideas resulting from the experiment are described in Chapter 6.2. An overview of question's mean grades is presented in Table 5.2.7.

Question	Mean
TM is easy to use (10 = Agree; 1 = Disagree)	9
TM is easy to learn (10 = Agree; 1 = Disagree)	9
Binding code artifacts to tasks directly in IDE reduces the effort needed to solve a task (10 = Agree; 1 = Disagree)	7.5
The related code artifacts list is useful for rapid navigation through the code (10 = Agree; 1 = Disagree)	9.5
TM helps developers to better and faster understand unfamiliar code (10 = Agree; 1 = Disagree)	6.5
TM helps developers to keep task specific elements together (10 = Agree; 1 = Disagree)	8.5
Generating tests helps developers to solve task (10 = Agree; 1 = Disagree)	6
Task completion should depend on satisfied tests (10 = Agree; 1 = Disagree)	7
TM helps developers to track the progress of the tasks (10 = Agree; 1 = Disagree)	7.25
TM helps developers to learn from previously solved tasks (10 = Agree; 1 = Disagree)	8
How satisfied were you with the TM ? (10 = High; 1= Low)	8
Total	**7.84**

Table 5.2.7 Questionnaire Results

5 Summary

5.1 Conclusions

In this document we presented the thesis that the task manager integrated into the IDE supports developers during programming. We identified problems which can occur during solving programming tasks. According to different kind of problems that can occur during developing a software, we introduced four types of task. Using the empirical experiment, we showed that the task manager can increase speed of programming. We also motivated our work to integrate our tool directly into the IDE to support a developer when navigating code artifacts. For this project we have chosen C# as the programming language and VS as the IDE. We developed the TM – a VS plug-in, which allows developers to define programming tasks directly in IDE and provides information about the software system at runtime. The tasks can be associated with code artifacts and to developers who are involved in the completion of the task. A binding between code artifacts and tasks can be done directly in the code view. When binding between a task and code artifact is done, TM generates a test class with template test cases for each code artifact. The test class has to be implemented by the developer in order to complete the task. TM detects when tests are satisfied and changes the status of a task to 'completed' or allows the developers to do that. By generating tests that have influence on the task completion, TM supports the idea of TDD. TM consists of the following parts:

- **Creating Tasks / Associating Code Artifacts** – By using TM we are able to define tasks and assign code artifacts to them. This can be done directly from within the IDE where tasks are being implemented.

- **Data Storage** – TM dynamically gathers information about code artifacts and stores the data in DB.

- **Enrichments** – TM provides new functionality in VS such as new windows to represent the current task, developers and code artifacts. TM registers also in VS new commands and shortcuts to control TM from within the IDE.

- **Visualizations / Navigation** – We visualize the associated code artifacts in a tree view which allows developers to directly navigate to the associated code artifact.

We evaluated our tool by conducting a controlled experiment with eight developers who are familiar with C #and VS. Developers were divided in two groups experimental and control group. The experimental group solved tasks with and the control group without TM. The results given by the experiment show that using TM meets developer's requirements. Our empirical experiment shows that TM reduces:

- 13 % time spent on working on task.
- 47 % time spent on navigating code artifacts.
- 27 % time needed to write tests.

In Chapter 4 we outlined the most important parts of the current implementation of TM. We can conclude that TM successfully supports developers during programming, improves communication exchange within the team and gives an overview of the tasks existing in the software system.

5.2 Perspectives

During the project we came up with new ideas how TM can be extended or improved. Most of these ideas are concerned with new features and were proposed by the developers participating in the evaluation.

Task Merging

Implementing possibility for merging tasks. The developer can select a completed task (source task) and a new one which is going to be implemented (target task). TM performs a forward merging to prepare templates for code artifacts in the target task. Templates are based on code artifacts contained in the source task. Task merging could decrease overhead when developing features similar to existing ones.

Hierarchy of Developers

An idea is to represent developers involved in the completion of a task in a hierarchy. Developers can have roles which determine their responsibilities. The hierarchy could give a better overview of the team and would clarify roles of each team member.

Limitation Removal

Limitations discussed in Chapter 3.4 can be removed if code artifacts would be additionally covered with annotations. An annotation would guarantee that even if the VS tools were not used to change a code artifact, the code artifact can be still found by TM.

Appendix A

Quick Start to TaskManager

A.1 Requirements

To be able to use TM, a standard installation of Visual Studio 2008 in any version and SQL Server preferably in version 2005, are needed. VS provides a distribution of SQL Server 2005 by default so it should be already available. Furthermore a current version of the C# and .NET framework should be installed on your system.

A.2 Installation

To install TM you need to start VS and add a new solution from subversion as follows:
File -> Subversion-> Open from Subversion…

Path to repository:
https://svn2.assembla.com/svn/taskmanager/

Put the file TaskManager.AddIn into your $Workspace\Addins folder. You are now able to compile TM. Once compiled TM is enabled and you can start working with it.

A.3 User's Guide

Detailed description of each feature and use cases can be found in Chapters 3.2 and 3.2.

Appendix B

Developer's Guide

As mentioned before we decided to use VS as a programming environment where we integrated and implemented TM. The motivation to use VS was that it offers many qualitative tutorials and feedback from Microsoft. Besides that VS proposes interesting libraries which can be used to easily produce a VS extension. From the number of programming languages supported by VS, we have chosen C# because of our programming experience with C# and the interest to enhance our knowledge about this language. The choice of IDE and programming language allows us to concentrate mostly on the functionality and final result of TM and not on implementation details.

B.1 Architecture

At the beginning of the project we identified two main parts:

- Development of a tool with all features described in chapter 3.2.
- Integration into IDE.

The first part could be quite complex so to decrease the complexity and the coupling between objects, the main program was split into three separate components: View-Logic-Data. Another advantage of dividing our tool into three parts is that code should become more readable. The relationships between components are shown in Fig. B.1.1.

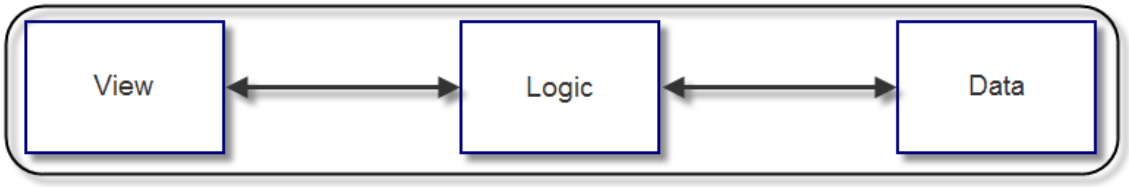

Fig. B.1.1 TM Components

If needed it is also possible to change one component and keep others without changes.

Data is responsible for saving, loading and managing data produced by TM. The most important part of the *Data* is the `ITableModule` interface which allows the *Logic* component to communicate with the DB using typed `DataSet`.

The most important component is *Logic*, which is responsible for the implementation of all features. It decides how to perform strategic actions, for example how to reflect and associate code artifacts with a task. The main part of that component is represented by `BusinessObject` (BO). BO represents an actor described in chapter 3.1. A BO has a number of actions that can be performed on that object, for example `TaskBO` with an edit method to change the existing task. The *Logic* component supports also communication between the *Data* and *View* components which are invisible to each other.

The *View* component is responsible for presentation. In our case we use `WindowsForm` to show the output. All actions are delegated to the *Logic* component, so adding new views like the tool window to the code view does not take too much effort. The *View* contains also *Cod* which plays the role of a helper component and contains classes with constant fields used as a text somewhere in the program.

The integration into IDE is done by creating a project of type Add-in which gives TM access to current instance of VS and so the possibility to extend VS and add a new functionality.

B.2 Plug-in Implementation

To extend VS it is necessary to create an Add-in project and implement `IDTExtensibility` and `IDTCommandTarget` interfaces [8] to define new behavior in VS. Add-in projects have access to instances of `DTE` objects. `DTE` can be understood as VS itself. With instance the developer is able to register new features in VS. The most important parts of the `DTE` Interface used by TM is shown in Fig.B.2.1. `IDTExtensibility` and `IDTCommandTarget` open the door to create something new in VS. TM uses that possibility by implementing three of many offered methods:

- `OnConnection()` – receives notification that an Add-in is being loaded. According to the observer pattern this method is used to register menu commands in VS and `EventHandler` for shortcuts, code artifact changes and solution changes. Used by TM `EventHandler` are marked with 'EH' in Fig B.2.1.
- `QueryStatus()`– This method is called when the command's availability is updated. In TM it is used to define how and when commands should be available and when not. Example: show menu entry only if a solution is open.
- `Exec()` – is called when the command is invoked. Used to specify the execution of TM commands. `Exec()` delegats immediately to the *Logic* component.

As shown in Fig. B.2.1 (marked with 'H') VS proposes the following hierarchy of source artifacts, beginning from the top `Solution`, with `Projects` which can contain many `CodeModel`'s they represents source files, which can contain `CodeElement`'s they represents classes and methods. TM allows to associate each element from this hierarchy with a task.

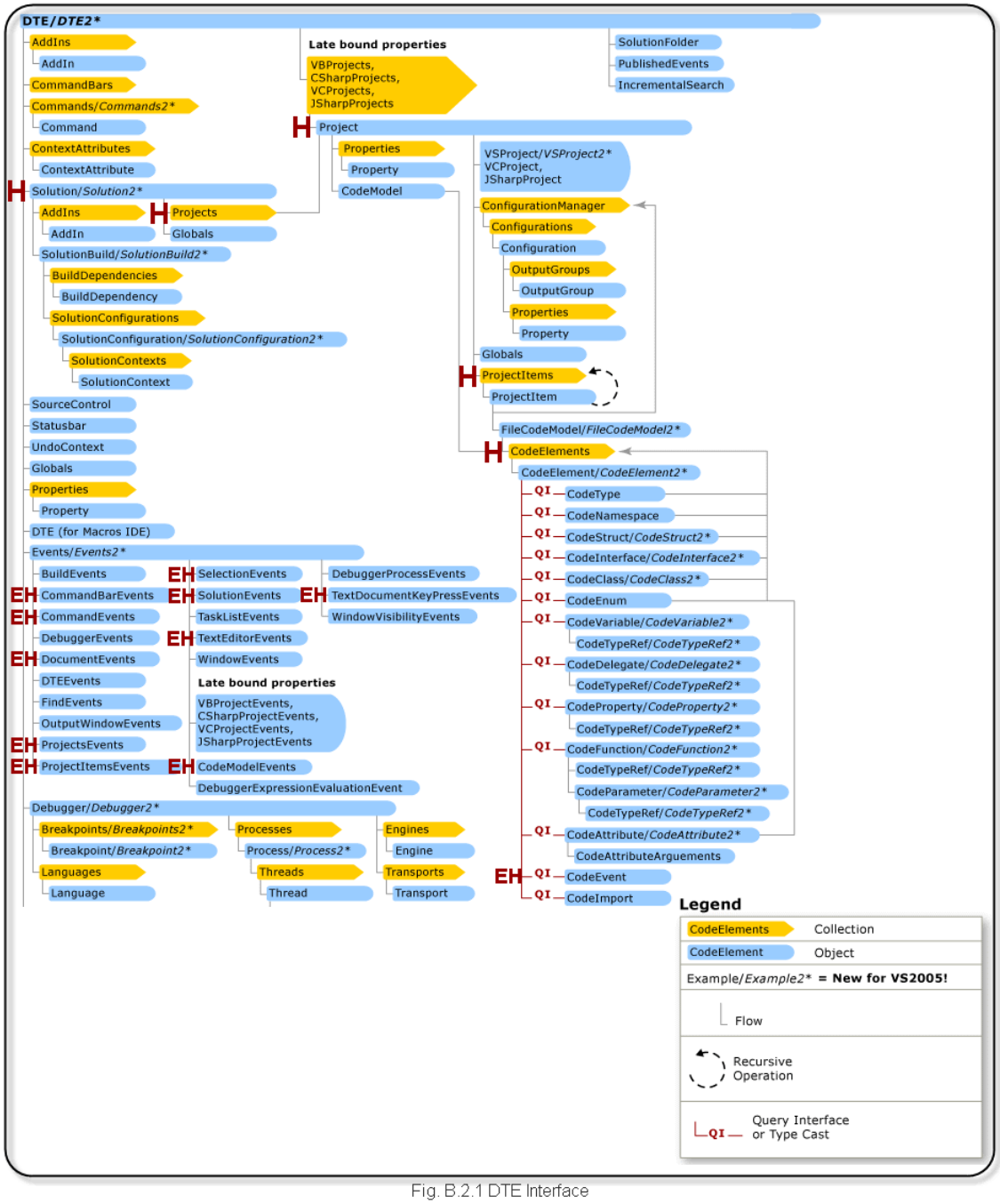

Fig. B.2.1 DTE Interface

B.3 Reflecting on Code Artifacts

A mechanism to observe and modify the structure and behavior of the developed program at runtime is required in two parts of that project. First to support a binding between a task and code artifacts, TM needs to know about all code artifacts contained in a solution and take an action if code artifacts change. Second, to generate and execute a task's test class with its test cases. This means that code artifacts have to be reflected dynamically every time when a developer wants to assign a code artifact to a task or check the progress of a task.

In the context of associating code artifacts to a task, TM uses the current instance of `DTE` object, which gives the access to the code artifacts contained in the solution and allows us to treat them as objects with their own methods and events. VS provides a mechanism to inform about changes on code artifacts by sending a notification to associated `EventHandlers`. A simple scenario here is when the developer renames the code artifact that is associated to a task. VS detects this change and sends the notification to the `TaskManagerEventHandler`, which then informs `TaskBO` that the DB needs to be updated. To catch notifications about changes on code artifacts TM adds new `EventHandlers` to `CodeEvent` and `ProjectEvent`, handling of events is delegated to the *Logic* component. The concept of event handling in TM is based on the observer pattern and is shown in Fig. B.3.1

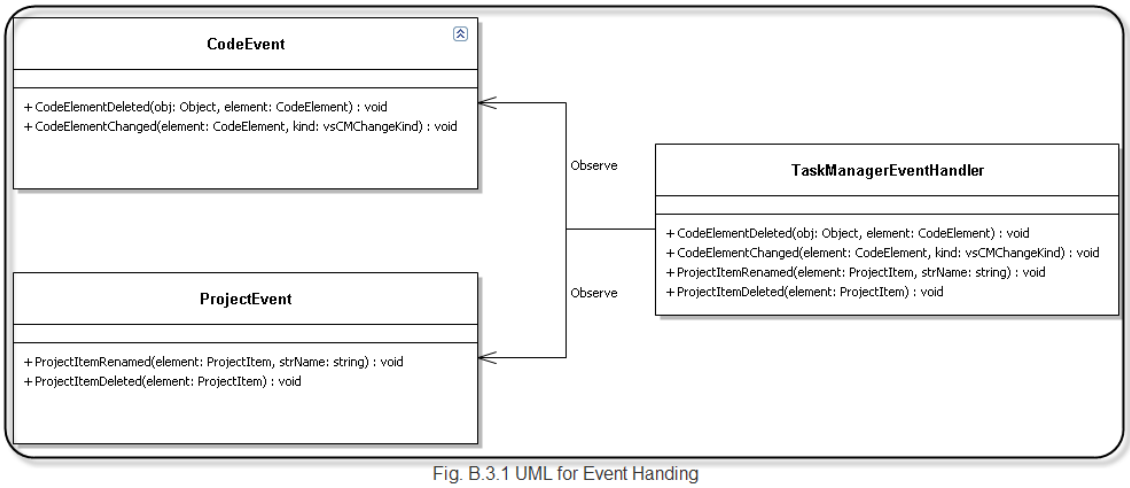

Fig. B.3.1 UML for Event Handing

One of the features offered by TM is test generation. A simple scenario here is when the developer creates a new task TM generates for this task a test class. TM provides also a mechanism to execute test cases contained in the generated test class. A sequence diagram representing how TM creates and executes tests is shown in Fig B.3.2

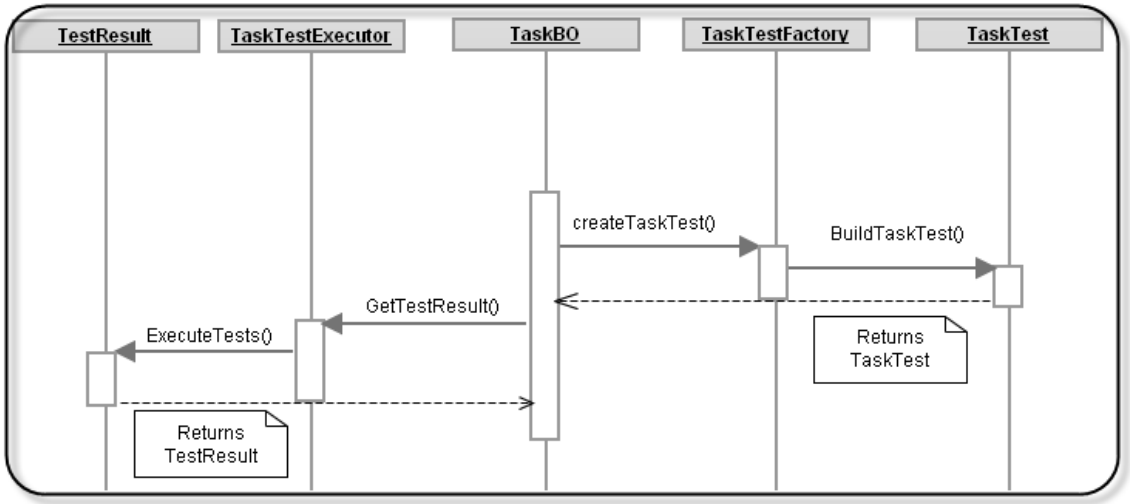

Fig. B.3.2 Sequence Diagram for Test Generation and Execution

In the context of test generation and execution of the task test class, TM uses `TaskTestFactory` and `TaskTestExecutor`. `TaskTestFactory` uses the

`CodeDom` library. Classes in this library can be used to model the structure of a source code document. The output of the source code document can be translated in an arbitrary programming language using the functionality provided by the *System.CodeDom.Compiler* library. `TaskTestFactory` takes a task and its assigned code artifacts to generate a new task's test class. The generated test class contains template test cases for each assigned to the task code artifact. Test cases are generated to fail after creation so the developer should implement them. To check how many tests are satisfied we use `TaskTestExecutor`. This class is responsible for creating reports about the satisfied test cases contained in a test class. `TaskTestExecutor` takes a test class and executes each test case contained in the test class. A UML diagram representing classes and relations between then is shown in Fig. B.3.3.

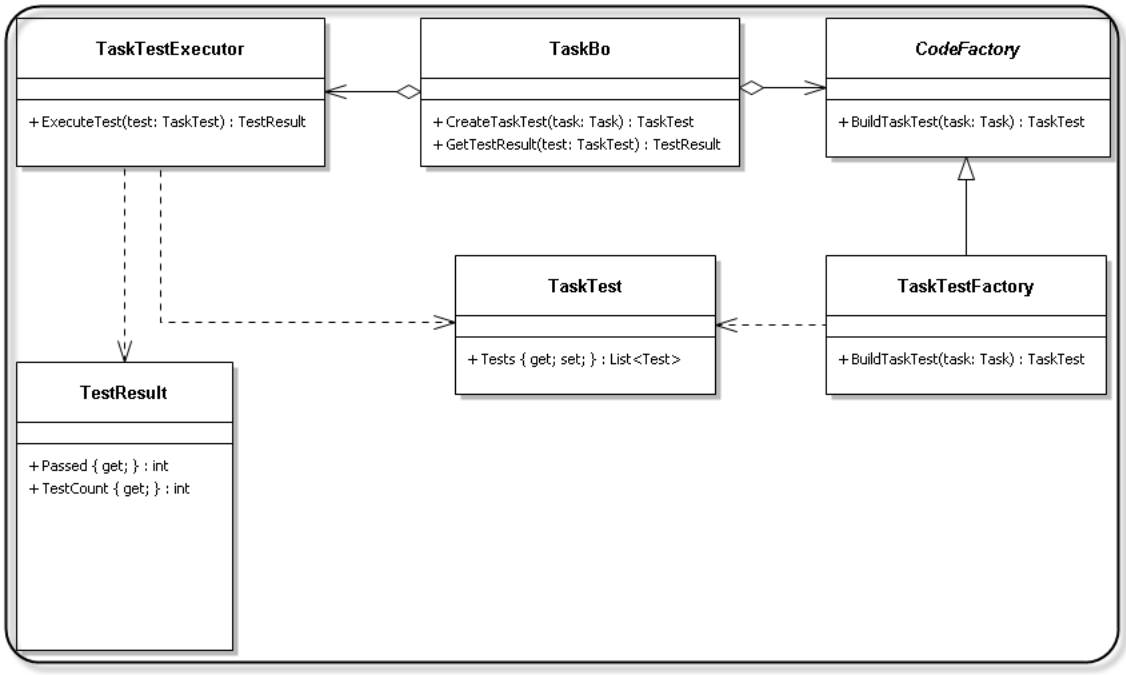

Fig. B.3.3 UML for Test Generation and Execution

B.4 Storing Task Data

TM stores all data in a DB, thus DB connection management is required. Developers are able to define a DB connection with two authentication modes. TM checks if the defined connection is valid. Developers can also choose the DB in which TM generates its tables. The best idea is to use the DB associated with a current solution, otherwise the DB needs to be created especially for TM. The reason of using a relational DB instead of a file based approach using XML, is to support concurrent access to data. Using a file based DB only one developer could do changes at a time, since one file can by checked out only by one developer at the same time. The access frequency to the DB is relatively high, so the usage of an optimistic lock pattern would lead to many conflicts, and a pessimistic lock pattern would block other developers from access to the file based DB. The database model used by TM is shown in Fig. B.4.1.

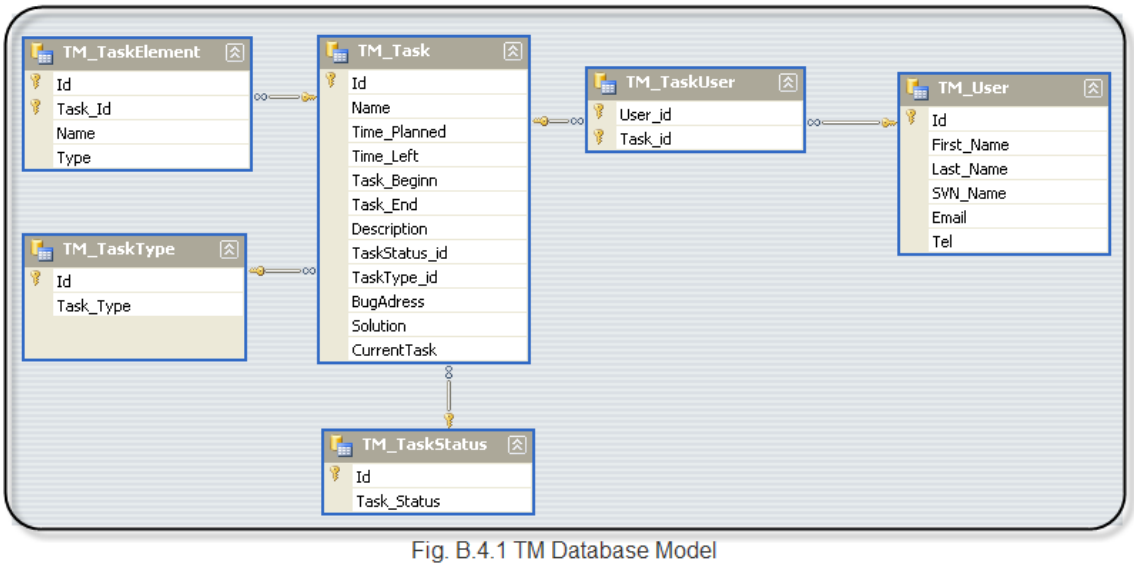

Fig. B.4.1 TM Database Model

Using *C#*, developers have to operate with `DataSet` which is a container of `DataTables`. In the standard solution offered by C#, each table needs to be downcast before operations like insert, update, select or delete can take place. To eliminate the requirement of casting we introduced an interface – `ITableModule` (*Data* component). The `ITableModule` interface allows us to perform simple operations like insert, update, fill table and delete on a typed `DataTable`. `ITableModule` with its base functionality is extended by specific child classes corresponding to each typed `DataTable`. `ITableModule` child classes support more complex select operations and provide a mechanism to fill specific report tables with data. A UML diagram is shown in Fig. B.4.2.

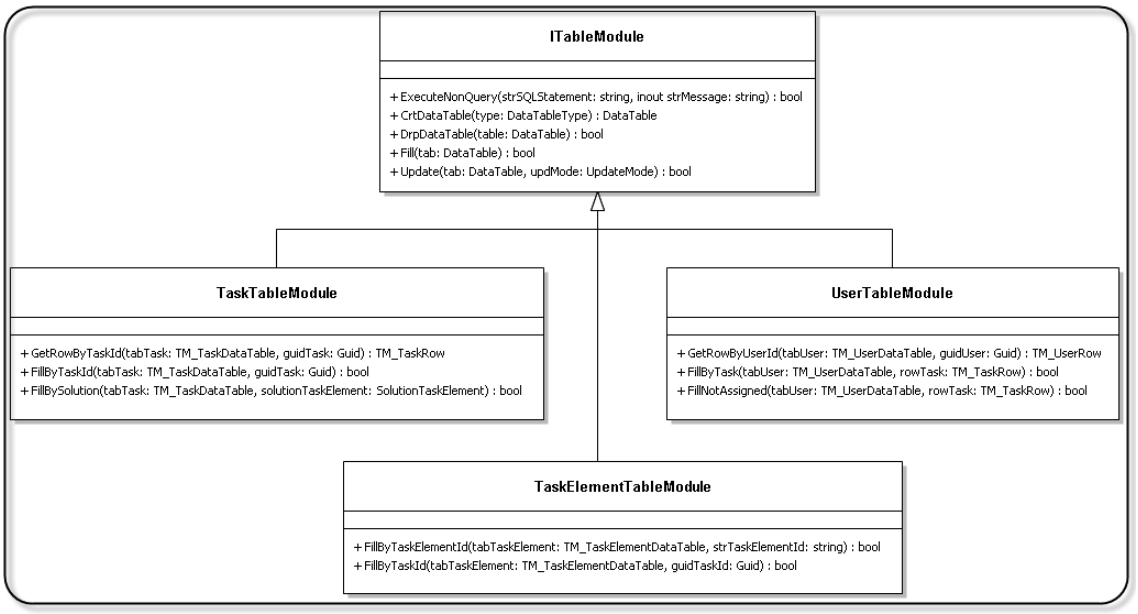

Fig. B.4.3. UML for DB management

TM stores task's references to code artifacts in the DB. TM performs an update on those references in the following situations:

- Each time a new code artifact is assigned to or deleted from a task.

- When associated code artifacts cannot be found within a solution while existing in the DB. The DB needs to be updated and the references to the orphaned elements have to be deleted.
- When a code artifact changes its name or path in a solution.

Bibliography

[1] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for IDEs. In AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, p. 159—168, ACM Press, New York, NY, USA, 2005.

[2] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, p. 1—11, ACM Press, New York, NY, USA, 2006.

[3] Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: Supporting Navigation in Software Maintenance. In International Conference on Software Maintenance (ICSM'05), p. 325—335, IEEE Computer Society, Washington, DC, USA, September 2005.

[4] Robert DeLine, Mary Czerwinski, and George G. Robertson. Easing Program Comprehension by Sharing Navigation Data. In VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, p. 241-248, IEEE Computer Society, Washington, DC, USA, 2005.

[5] Bellotti, V., Dalal, B., Good, N., Bobrow, D. G., Ducheneaut, N. What a to-do: studies of task management towards the design of a personal task list manager. Proceedings of the Conference on Human Factors in Computing Systems. p. 735-742, 2004.

[6] Murphy, G., Kersten, M., Robillard, M. and Cubranic, D. The Emergent Structure of Development Tasks. Proceedings of the European Conference on Object-Oriented Programming. p. 33-48, 2005.

[7] M. Robillard and G. Murphy, "FEAT: A tool for locating, describing, and analyzing concerns in source code," Proceedings of 25th International Conference on Software Engineering, May 2003.

[8] Keyvan Nayyeri, “ProfessionalVisual Studio® 2008 Extensibility”, Wiley Publishing, IN 46256, ISBN: 978-0-470-23084-8, 2007