Functional Specification

VHS-AM Project

Functional Specification

Version 1.1

Document History

Date	Version	Description	Author
11/08/2003	0.1	Document creation	André Furtado
22/08/2003	0.2	Elaboration of “essential” use cases	André Furtado
26/12/2003	0.3	Revision of use cases UC03 and UC04	Mauro Araújo
29/12/2003	0.4	Partial translation to English	André Furtado
06/12/2004	1.0	Final translation to English	André Furtado
22/12/2004	1.1	Split of some use cases	André Furtado

Table of Contents

1. Introduction

This document specifies the use cases and non-functional requirements of the project VHS-AM, aiming at providing as much information as possible for developers, allowing designing and implementation activities, and also for testers, allowing the evolution of test approaches into detailed planning and designing.

1.1 Document conventions, definitions and abbreviations

The correct understanding of this document requires familiarity with some specific definitions, conventions and abbreviations, as described below.

1.1.1 Use cases and non-functional requirements identifiers

Each use case is prefaced by a unique identifier. The identification has the form [UCXX], where XX is an integer number starting with 01 for the first use case and is incremented by one for each additional use case. Non-functional requirements, on the other hand, start with [NFR01], but follow the same incremental rule.

Secondary use case flows also have identifiers. If the secondary flow is an alternate flow, the form of the identifier is [AFXX], where XX ranges from 01 to the number of alternate flows of the use case. Identifier for error flows, on the other hand, follows the form [EFXX], where XX once again starts with 01 and is incremented by each error flow. In both flows, XX is reset to 01 for each use case.

When it is necessary to reference a specific use case flow, the following form is used: [UCXX].[AFXX] for alternate flows and [UCXX].[EFXX] for error flows.

1.1.2 Use cases and non-functional requirements priorities

To establish use cases (UC) and non-functional requirements (NFRs) priorities, in sections 3 and 4, the following classification is used:

Essential: All essential UCs/NFRs must be satisfied in order for a solution to be released. Such UCs/NFRs are indispensable; therefore, they must be implemented.

Important: The lack of an important UC/NFR in a solution will not block its release. However, the application will not be classified as completely satisfactory.

Desirable: This kind of UC/NFR does not compromise solutions features, i.e., a solution can be deployed satisfactorily without desirable UCs/NFRs. These are the first UCs/NFRs to be postponed to next versions when the project runs out of time or resources.

2. Actors

This section describes the actors that will interact with the solution. Actors can be specific users of the application, external hardware/software devices or even other applications that exchange information with the solution to be developed.

Actor	Description
Haskell Programmer	Individual that will use the extended Visual Studio .NET to create Haskell applications.
Assignment Manager Teacher	Individual that will use Assignment Manager (improved with the extended VS.NET) with the purpose to create

	and advertise Haskell coding exercises.
Assignment Manager Student	Individual that will use Assignment Manager (improved with the extended VS.NET) with the purpose to solve and submit Haskell coding exercises.
GHC (Glasgow Haskell Compiler)	A Haskell compiler eventually invoked by VS.NET.
GHCi (Glasgow Haskell Compiler Interpreter)	A Haskell interpreter eventually invoked by VS.NET.

3. Use Cases

This section presents all solution use cases.

[UC01] To develop Haskell code with editor support

Priority:	☒	Essential	☐	Important	☐	Desirable
Actors:	Haskell Programmer					
Related use cases:						

Description: When coding in Haskell, VS.NET should provide visual facilities that support the implementation process, in order to improve programmers' productivity. Actually, this use case is the result composition of six "sub use cases":

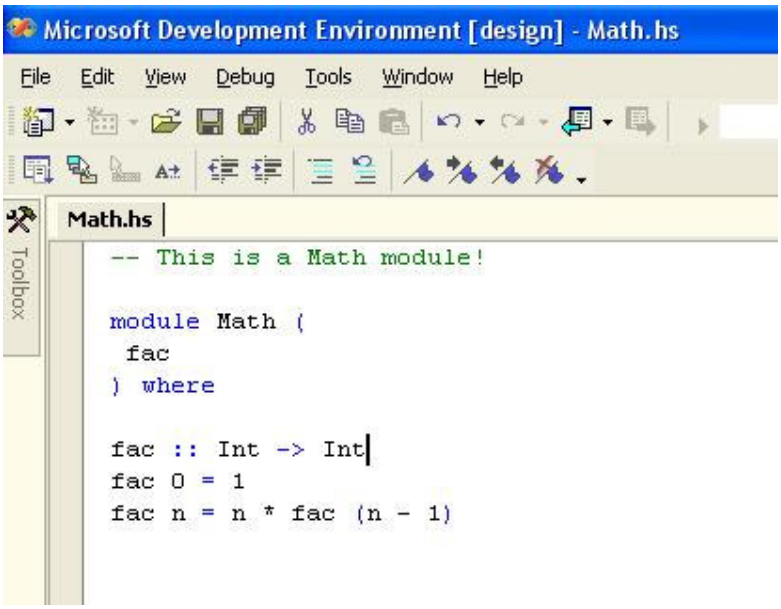
[UC01A] Syntax coloring: different colors are associated to each kind of code element, such as Haskell keywords, comments and operators, for example.

[UC01B] Brace Matching: when a parenthesis, a brace or a curly brace is closed, the enclosing text is emphasized, leading to a clearer identification of the contents of the block.

IntelliSense: when the programmer triggers context-specific events, some context-sensitive features are executed. Examples of such event/feature pairs follow below, in priority order:

- **[UC01C] Quick info:** the type of an identifier is shown by "mouse-overing" it
- **[UC01D] Auto-complete:** when a specific key combination is pressed, (such as CTRL + space bar) or when a popup list member is select (see below), the "half-word" in which the cursor is located is completed.
- **[UC01E] Parameter-info:** when a specific key combination is pressed (such as CTRL + SHFT + space bar) just after a function name, its return type and its parameters types are shown.
- **[UC01F] Member list popup:** when a specific key combination is pressed (such as CTRL + J) or when a member-access operator is typed, a popup list containing all possible identifiers is shown.

A user interface draft for this use case is illustrated below, in Picture 1:



Picture 1 – GUI draft: Haskell code editor support

Inputs and pre-conditions: Appropriate VSPackages must be registered; a Haskell module file must be active in VS.NET.

Outputs and post-conditions: VS.NET reacts to user input, activating corresponding features such syntax coloring, brace matching or IntelliSense.

Main Flow

1. The Haskell programmer types some code, triggering specific events;
2. VS.NET visually reacts to the triggered events.

Alternative Flows

[AF01] Ambiguous auto-complete

If VS.NET is requested to provide an auto-complete and more than one option is available for completing the "half-word", a member list popup should be displayed, containing all available options.

[AF02] Member list popup canceling

If the programmer presses ESC while a member list popup is displayed or clicks anywhere outside it, the member list popup should disappear.

Error Flows

None.

[UC02] To create a new *Haskell Console Application* project

Priority:	☒	Essential	☐	Important	☐	Desirable
Actors:	Haskell Programmer					

Related use cases:	

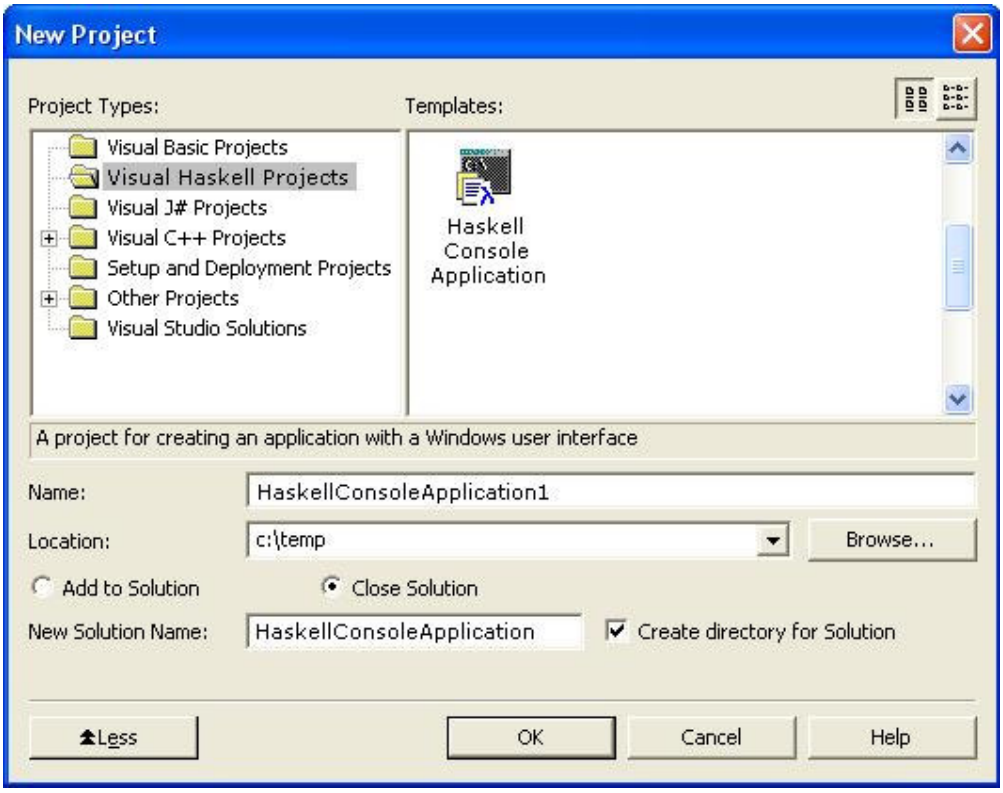
Description: Besides just editing Haskell code, programmers should be able to logically group Haskell modules into a project (which is, obviously, part of a VS.NET solution). This is required to compile and execute the Haskell code and to allow VS.NET integration with AM.

Inputs and pre-conditions: Appropriate VSPackages must be registered; VS.NET 2003 must be running.

Outputs and post-conditions: A *Haskell Console Application* project is created, containing a Haskell module template.

Main Flow

1. The programmer access the menu <File> → <New> → <Project>;
2. VS.NET requests the name and type of project, as illustrated in Picture 2



Picture 2 – GUI draft: choosing VS.NET project types

3. The programmer chooses the *Haskell Console Application* project type and defines its name and location (VS.NET itself manages the existence of an already opened solution, in order to request additional input to the programmer);
4. The programmer clicks the <OK> button;
5. A new project is created and displayed, containing a Haskell module template.

Alternative Flows

[AF01] Operation canceling

If the programmer clicks the <Cancel> button (see Picture 2), the <New Project> dialog closes and no project is created.

Error Flows

[EF01] Invalid project name or location

If the programmer enters an invalid project name or location, in the <New Project> dialog (see Picture 2), a friendly error message is displayed and no project is created.

[UC03] To create and advertise Haskell exercises through Assignment Manager

Priority:	☒	Essential	☐	Important	☐	Desirable
Actors:	Assignment Manager Teacher					
Related use cases:	UC02					

Description: The AM Teacher should be capable of using Haskell as well as any other language originally supported by VS.NET in order to create and advertise Haskell coding exercises to students. These exercises may contain hidden code that should be filled by students.

Inputs and pre-conditions: Appropriate VSPackages must be registered; AM Server must be running; a *Haskell Console Application* project should have been created; a course should exist in AM Server; AM Faculty Client should be installed and running in the teacher computer; the teacher should have been registered in AM Server as a role capable of advertising exercises.

Outputs and post-conditions: A Haskell exercise (with or without hidden code) is advertised.

Main Flow

1. Once running VS.NET 2003, the teacher opens a *Haskell Console Application* project;
2. The teacher uses its username and password to logon in AM Server;
3. The teacher advertises the project as an exercise, following AM advertisement procedures;
4. AM makes the advertised exercise available to the students.

Alternative Flows

[AF01] Student code indication

Before advertising an exercise, the teacher selects pieces of code mark it as “student code”. This makes these pieces of code to be hidden when the exercise is advertised.

Error Flows

None.

[UC04] To solve and submit Haskell exercises through Assignment Manager

Priority:	☒	Essential	☐	Important	☐	Desirable
Actors:	Assignment Manager Student					
Related use cases:	UC02 e UC03					

Description: As well as AM Teachers, AM Students must be able to work with Haskell in the same way they do with other language originally supported by VS.NET, taking advantage from the extended IDE to solve advertised exercises before submitting them back through AM.

Inputs and pre-conditions: Appropriate VSPackages must be registered; AM Server must be running; a Haskell exercise should have been advertised by an AM Teacher; AM Student Client should be installed and running in the student computer; the student should have been registered in AM Server as a "student role" of the related course.

Outputs and post-conditions: The advertised exercise is solved and submitted back to the teacher, through Assignment Manager.

Main Flow

1. Once running VS.NET, the student uses its username and password to logon in AM Server;
2. The student access the advertised Haskell exercise and solves it;
3. The student submits the solved exercise back to the teacher, through AM student client.

Alternative Flows

[AF01] Student code filling

In step 2, if the exercise contains hidden code, the student should complete it before submitting.

Error Flows

None.

[UC05] To add a new Haskell module to a *Haskell Console Application* project

Priority:	☒	Essential	☐	Important	☐	Desirable
Actors:	Haskell Programmer					
Related use cases:	UC02					

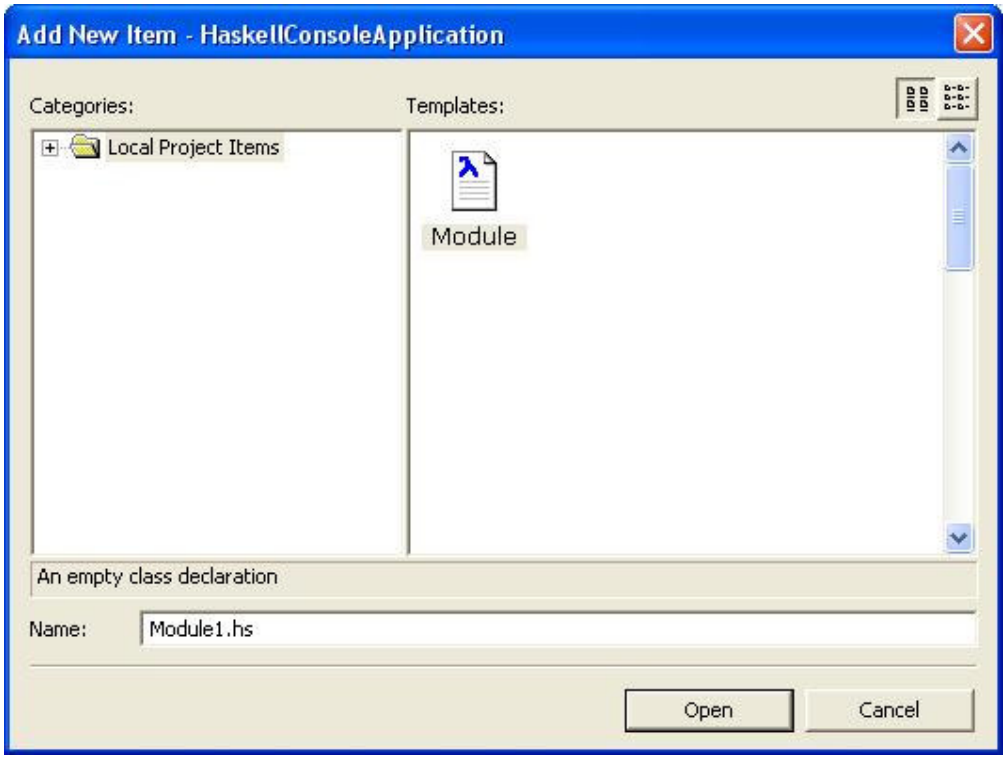
Description: A Haskell project cannot be restricted to a single Haskell module. It must be possible for Haskell Programmers to add new Haskell modules as they wish.

Inputs and pre-conditions: Appropriate VSPackages must be registered; a *Haskell Console Application* project must be active in VS.NET.

Outputs and post-conditions: The new Haskell module is added to the project.

Main flow

1. In solution explorer, the programmer clicks with the right mouse button in the project to which a new Haskell module will be added;
2. The programmer clicks <Add> → <New Module>;
3. VS.NET displays a window where the programmer provides the new module name, as shown in Picture 3.



Picture 3 – GUI draft: new Haskell module

4. The programmer clicks <OK>;
5. A Haskell module template, containing the name provided by the programmer, is added to the project.

Alternative Flows

[AF01] Operation canceling

If the programmer clicks <Cancel> in the window shown in Picture 3, no new Haskell module is added to the project.

Error flows

[EF01] Invalid Haskell module name

If the programmer provides an invalid module name in the window shown in Picture 3, a friendly error message is displayed and no new Haskell module is added to the project.

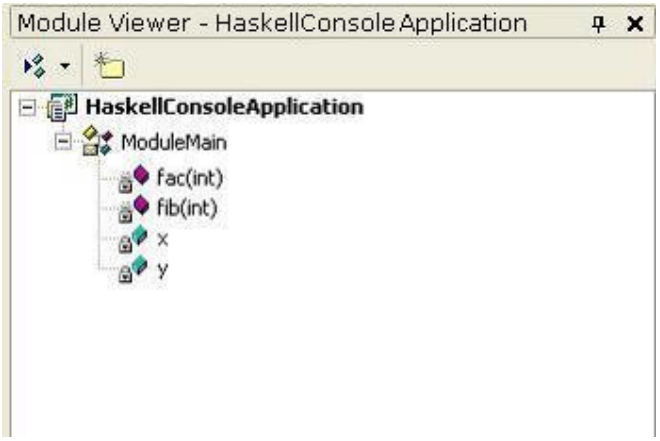
[UC06] To navigate through a Haskell module viewer

Priority:	☒	Essential	☐	Important	☐	Desirable
Actors:	Haskell Programmer					
Related use cases:	UC02					

Description: In order for programmers to better understand the structure of the code they are developing, a module viewer should be present to show the hierarchical relation between the code elements, such as modules, functions, classes and data types.

Inputs and pre-conditions: Appropriate VSPackages must be registered; a *Haskell Console Application* project must be active in VS.NET.

Outputs and post-conditions: A tool window containing a navigable hierarchical structure of the code displays the project modules, function, classes and data types, as shown in Picture 4.



Picture 4 – GUI draft: Module Viewer

Main Flow

1. The programmer requests the visualization of the Module Viewer;
2. VS.NET displays a tool window containing a navigable hierarchical structure of the code;
3. The programmer navigates through the code hierarchy.

Alternative Flows

None.

Error Flows

None.

[UC07] To dynamically obtain error messages

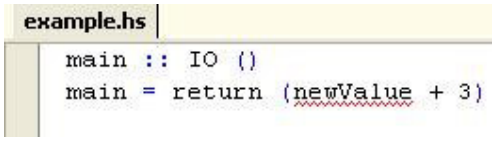
Priority:	Depends on the error type (see below)
Actors:	Haskell Programmer
Related use cases:	

Description: While programmers are developing code, the awareness of errors should not be postponed to the building phase. In order to improve programmers' productiveness, errors should be identified as soon as possible. Two error types is covered by this use case:

[UC07A]: Lexical/Syntax errors (*Priority: essential*)

[UC07B]: Semantic errors (*Priority: important*)

The errors dynamically identified by the first version of this solution will be restricted to lexical/parser errors, underlined with a red color in VS.NET text editor, as shown in Picture 5.



Picture 5 - GUI draft: dynamic error messages

Inputs and pre-conditions: Appropriate VSPackages must be registered; a Haskell module file must be active in VS.NET.

Outputs and post-conditions: VS.NET underlines the part of the code containing lexical/parser errors.

Main flow

1. The programmer types code with lexical/parser errors in VS.NET text editor;
2. VS.NET visually reacts to the errors by red-underlining the part of the code with errors.

Alternative Flows

[AF01] Error correction

As soon as the programmer corrects the error, the red underline should disappear.

Error flows

None.

[UC08] To view error messages in taskbar

Priority:	☐	Essential	☒	Important	☐	Desirable
Actors:	Haskell Programmer					

Related use cases:	

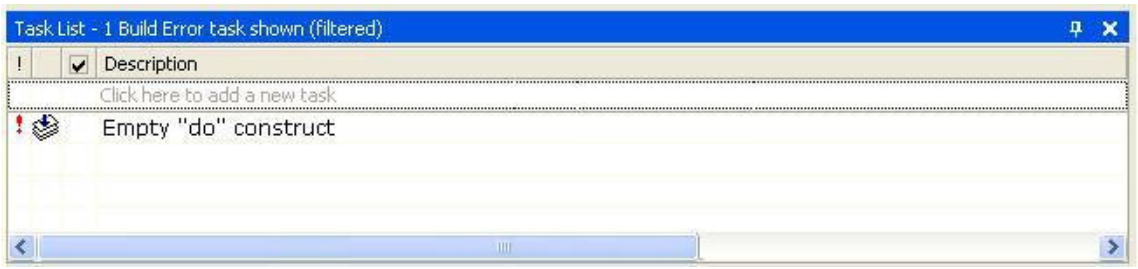
Description: The programmer should be able to quickly visualize all of the current errors of the application under development. In VS.NET, the taskbar is the place where all error messages are stick together. Error messages related to the errors dynamically identified should be added to the taskbar as well as new errors that arise after a failed compilation.

Inputs and pre-conditions: Appropriate VSPackages must be registered; a Haskell module file must be active in VS.NET.

Outputs and post-conditions: The taskbar displays the error messages.

Main flow

1. The programmer types code with lexical/parser errors in VS.NET edit window;
2. VS.NET visually reacts to the errors by adding them to the taskbar, as shown in Picture 6.



Picture 6 - GUI Draft: errors in the taskbar

Alternative Flows

[AF01] Post-compilation errors

If the programmer requests the VS.NET solution under development to be compiled and the compilation fails, new error messages related to the failure are added to the taskbar.

Error flows

None.

[UC09] To go to source code location from taskbar error messages

Priority:	☐	Essential	☒	Important	☐	Desirable
Actors:	Haskell Programmer					
Related use cases:	UC08					

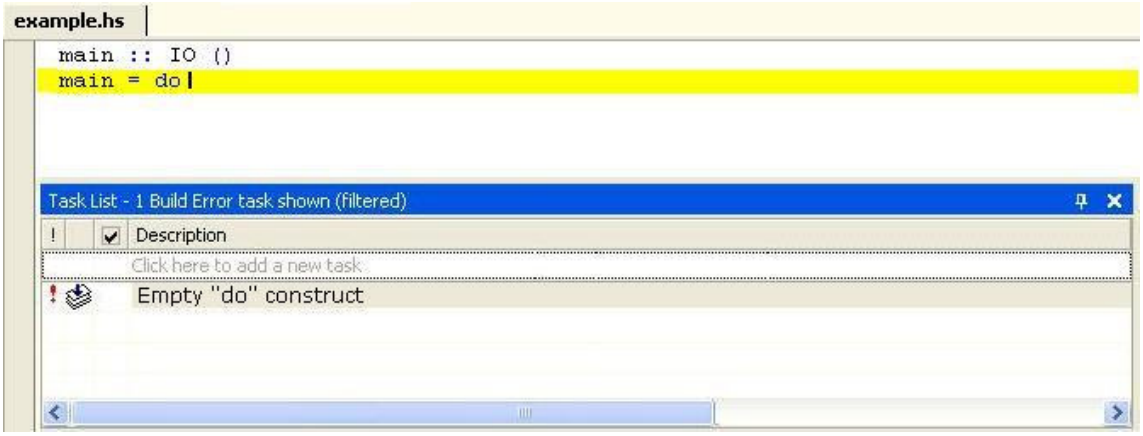
Description: In order to enhance programmers' productiveness, double-clicking error messages in the taskbar should direct the cursor to the appropriate source code location containing the error.

Inputs and pre-conditions: Appropriate VSPackages must be registered; a Haskell module file must be active in VS.NET; at least one error message should be displayed in the taskbar.

Outputs and post-conditions: The input cursor is moved to the code line that originated the error and this code line is highlighted.

Main flow

1. The programmer double-clicks an error message in the taskbar;
2. VS.NET highlights the line that originated the corresponding error and directs the input cursor to there, as shown in Picture 7.



Picture 7 - GUI draft: linking error messages to code

Alternative Flows

None.

Error flows

None.

[UC10] To jump to binding site of an identifier ("go to definition")

Priority:	☐	Essential	☒	Important	☐	Desirable
Actors:	Haskell Programmer					
Related use cases:						

Description: The programmer should be able to quickly find where an identifier was declared.

Inputs and pre-conditions: Appropriate VSPackages must be registered; a Haskell module file must be active in VS.NET.

Outputs and post-conditions: The input cursor is moved to the code line where the identifier is declared.

Main flow

1. In the text editor, the programmer clicks in an identifier with the right mouse button;
2. VS.NET displays a popup menu;
3. The programmer clicks the option <Go to definition> in the popup menu;
4. VS.NET moves the input cursor to the code line where the identifier is declared.

Alternative Flows

None.

Error flows

None.

[UC11] To add comments (*TODO*/*HACK*/*UNDONE*) to the taskbar

Priority:	☐	Essential	☐	Important	☒	Desirable
Actors:	Programador Haskell					
Related use cases:						

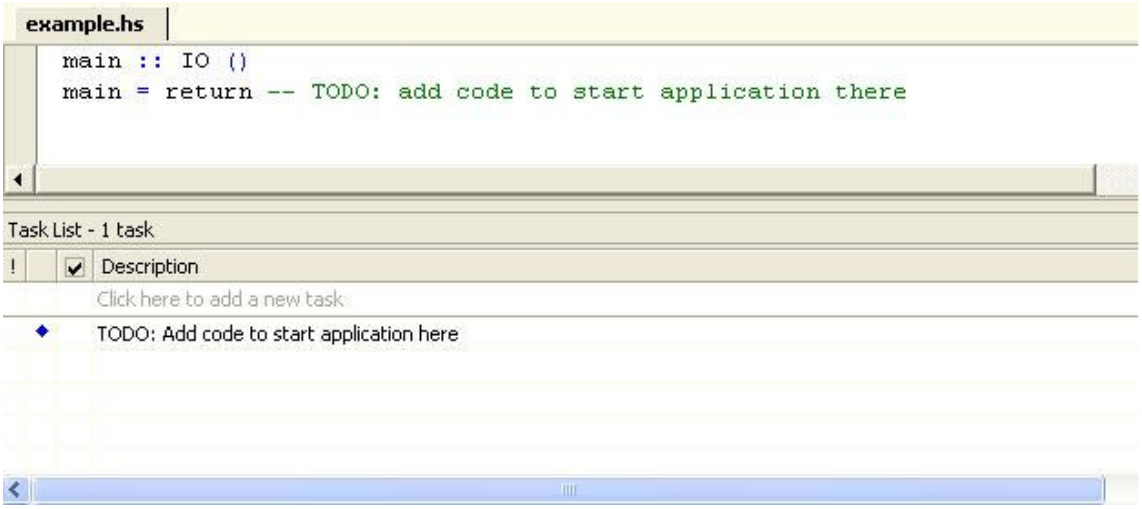
Description: Some programmers might find useful to leave special comments in the code to remind themselves or others of works that still needs to be done (or need some kind of special attention). VS.NET default special comments are TODO, HACK and UNDONE. Haskell projects should be able to support such comments.

Inputs and pre-conditions: Appropriate VSPackages must be registered; a *Haskell Console Application* project must be active in VS.NET.

Outputs and post-conditions: TODO, HACK and/or UNDONE special comments are added to the taskbar.

Main flow

1. The programmer types a line comment starting with TODO, HACK or UNDONE;
2. VS.NET recognizes the special comment and adds it to the task bar, as shown in Picture 8.



Picture 8 - GUI draft: special comments in taskbar

Alternative Flows

[AF01] Special comment deletion

If a TODO, HACK or UNDONE comment is deleted from the source code, its corresponding entrance in the task bar should also be deleted.

Error flows

None.

[UC12] To use a Haskell *navigation bar*

Priority:	☐	Essential	☐	Important	☒	Desirable
Actors:	Haskell Programmer					
Related use cases:						

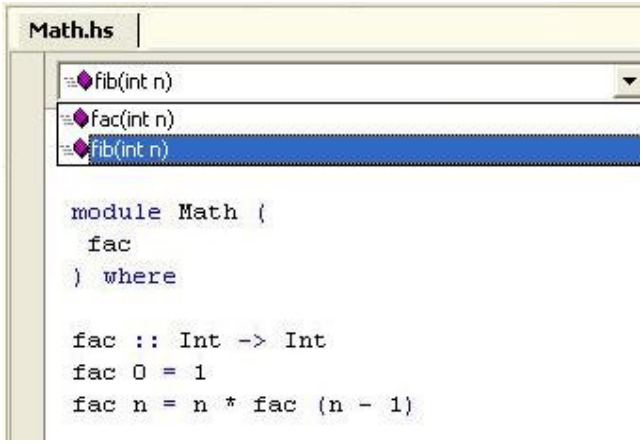
Description: The navigation bar is a popup menu that lists the functions of the currently active Haskell module, in order for the programmer to find them more quickly.

Inputs and pre-conditions: Appropriate VSPackages must be registered; a Haskell module file must be active in VS.NET.

Outputs and post-conditions: The input cursor is moved to the code line where the selected function is declared.

Main flow

1. The programmer picks a function from the popup list of the navigation bar, as shown in Picture 9;
2. VS.NET moves the input cursor to the code line where the identifier is declared.



Picture 9 - GUI draft: navigation bar

[UC13] To interactively query the Haskell program through GHCi

Priority:	☐	Essential	☐	Important	☒	Desirable
Actors:	Haskell Programmer and GHCi					
Related use cases:						

Description: The programmer should be able to query to the Haskell code under development, without having to compile it. This may be useful for unit testing, for example.

Inputs and pre-conditions: Appropriate VSPackages must be registered; a *Haskell Console Application* project must be active.

Outputs and post-conditions: The result of the query is displayed in a tool window.

Main flow

1. The programmer access the query window;
2. The programmer types the query string in the query window;
3. VS.NET invokes GHCi to process the query;
4. VS.NET displays the result from GHCi in the query window, as shown in Picture 10;

Picture 10 - GUI draft: GHCi interactive query window

Alternative Flows

None.

Error flows

[EF01] Invalid query string

If the provided query string is invalid, VS.NET displays a friendly error message in the interactive query window.

[UC14] To install/register VSPackages

Priority:	☒	Essential	☐	Important	☐	Desirable
Actors:	Haskell Programmer, Assignment Manager Student and Assignment Manager Teacher					
Related use cases:						

Description: This use case is related to the deployment, in VS.NET 2003, of the VSPackages developed in this solution.

Inputs and pre-conditions: VS.NET 2003 must have been installed in the target machine; appropriate VSPackages (dlls) must have been developed and tested; a license key must have been acquired for the VSPackages.

Outputs and post-conditions: VSPackages are registered and recognized by VS.NET 2003.

Main Flow

1. The user register the VSPackages using the acquired license;
2. VS.NET 2003 loads the VSPackages on demand, answering to requests to their functionalities.

Alternative Flows

None.

Error Flows

None.

[UC15] To compile a Haskell application

Priority:	☒	Essential	☐	Important	☐	Desirable
Actors:	Haskell Programmer, Assignment Manager Student, Assignment Manager Teacher and GHC					
Related use cases:						

Description: Haskell programmers, assignment manager students and teachers should be able to compile the VS.NET solution under development, in order to build an executable file and check if the solution is free of compilation errors.

Inputs and pre-conditions: Appropriate VSPackages must be registered; a *Haskell Console Application* project must be active in VS.NET; GHC must have been installed.

Outputs and post-conditions: Object and executable files are generated by GHC.

Main Flow

1. The programmer requests the VS.NET solution under development to be compiled;
2. VS.NET invokes GHC, feeding it with the information required to compile the VS.NET solution;
3. Output files resulting from compilation are generated by GHC.

Alternative Flows

None.

Error Flows

[EF01] Compiling error

If the *Haskell Console Application* is not compiling, messages related to the compiling errors, possibly extracted by GHC, must be displayed in a friendly manner (such as in the taskbar). No executable files are generated.

[UC16] To run a Haskell Application

Priority:	☒	Essential	☐	Important	☐	Desirable
Actors:	Haskell Programmer, Assignment Manager Student, Assignment Manager Teacher and GHC					
Related use cases:	UC15					

Description: Haskell programmers, assignment manager students and teachers should be able to run a VS.NET solution, containing a Haskell project, in order to interact with it to look for errors and to measure its quality,

Inputs and pre-conditions: Appropriate VSPackages must be registered; a *Haskell Console Application* project must be active in VS.NET.

Outputs and post-conditions: All VS.NET solution files are saved and the application is executed.

Main Flow

1. The programmer requests the application to run;
2. The executable file resulting from the last solution compilation is executed.

Alternative Flows

[AF01] Solution compiling

If the VS.NET solution was not compiled since the last change, VS.NET saves all its files and compiles it, invoking GHC. This happens between steps 1 and 2 above.

Error Flows

[EF01] Compiling error

If the programmer requests the running of a solution which contains a *Haskell Console Application* project that is not compiling, compiling error messages, possibly extracted from GHC, are displayed in a friendly manner. The application under development is not executed.

4. Non-functional Requirements

This section presents all non-functional requirements of the VHS-AM project. These non-functional requirements are related to different issues such as usability, reliability, performance, deployment, pattern compliance and software/hardware requirements.

[NFR01] VSIT Compliance

VSIT (Visual Studio .NET Integration Test), developed by Microsoft, is a suite for conducting integration testing. The VSIT suite verifies that a VSPackage does not break any major functionality in Visual Studio. The primary objective of these tests is to ensure that installing or uninstalling VSPackages works as expected with Visual Studio.

Priority:	☒	Essential	☐	Important	☐	Desirable
Related use cases:						

[NFR02] GHC lexer/parser integration

The lexer/parser to be used by the application must be the same lexer/parser used by GHC. This will support some important solution features, such as a better extensibility of the solution in relation with eventual Haskell evolutions and a better way of extracting semantic information from the code developed by the programmer.

Priority:	☐	Essential	☒	Important	☐	Desirable
Related use cases:						

[NFR03] Usability

Once all developed VSPackages are installed and registered, it should be not necessary for VS.NET and AM users to access or modify any special configuration of these tools in order to use them with Haskell. In other words, users must fell as they were using any language originally supported by VS.NET, such as C# or VB.NET.

Priority:	☐	Essential	☒	Important	☐	Desirable
Related use cases:						

[NFR04] Stability

When using the extended VS.NET and AM with Haskell, users must notice that the applications behave as stable as when other language originally supported by VS.NET, such as C# or VB.NET, is used.

Priority:	☒	Essential	☐	Important	☐	Desirable
Related use cases:						

[NFR05] Performance

When using the extended VS.NET and AM with Haskell, users must not notice any performance drawback in relation to the use of any language originally supported by VS.NET, such as C# and VB.NET. However, some delay may be expected while VSPackages are loaded.

Priority:	☐	Essential	☒	Important	☐	Desirable
Related use cases:						

[NFR06] Deployment

The deployment process of the application, including the registering of the developed VSPackages, must be an intuitive process and should not require an advance experience from users. Users must not be required to do any special deployment steps other than the usual steps in deploying/registering common VSPackages. An installation guide must be developed in order to provide a step-by-step guidance to the solution deployment process, referencing a more comprehensive bibliography. This installation guide should also describe specific requirements for the deployment, such as the presence of GHC and the .NET Framework, for example.

Priority:	☐	Essential	☒	Important	☐	Desirable
Related use cases:						

[NFR07] Software

The application must run under Visual Studio .NET 2003. The previous version of VS.NET (2002) is not part of the solution scope.

Priority:	☒	Essential	☐	Important	☐	Desirable
Related use cases:						

[NFR08] Hardware

The application should not require any special hardware configuration other than the original configuration required for VS.NET and AM.

Priority:	☐	Essential	☒	Important	☐	Desirable
Related use cases:						

5. Analysis & Design Artifacts

This section references the artifacts that compile the information generated by the Analysis & Design discipline.

Conceptual / Logical / Physical diagrams	*<not ready yet>*
Installing requirements diagram	*<not ready yet>*

6. References

- Visual Studio .NET Integration Program Documentation (comes along with VSIP and can be accessed through VS.NET Help);
- Griffiths I., Flanders J., Sells C. **Mastering Visual Studio .NET**, O'Reilly, 1st edition, March 2003.