**Task Module**

Architectural Overview [Eli]

ConGolog Task Module library consists of several modules:

*ConGolog Compiler* - Compiles the ConGolog based script to a Task object. Each Task is a representation of the task structure.

*Recorder*- Records and stores all the activities that were reported to the library. This module also manages and maintains the active experiments in the system.

*Task* – Represents a specific usability scenario that was specified in ConGolog script and has been created by compiling a ConGolog script by the TML compiler.

*Experiment* – Collection of tasks. Each experiment is independently responsible for managing its tasks (each task can belong to several different experiments). Self manageable – can be independently enabled and disabled.

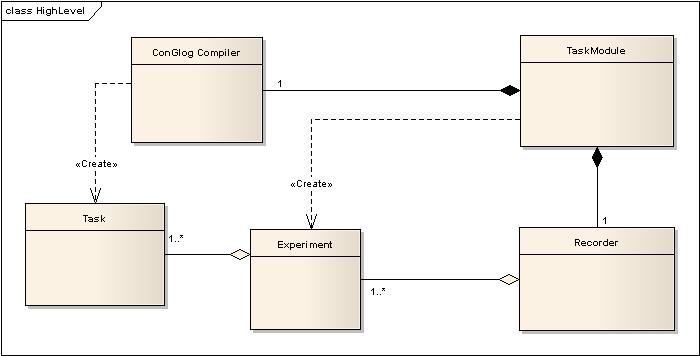


Figure 1 – High level design overview

Functionality Overview [Assaf]

In-order for a program to support the *Task Module Library (TML)*, developers needs to “wire” their programs in “interesting” locations throughout their code. The *Task Module* *Library*, leave it to the developer to define those points of interest (e.g. method invocations object state changes or curtain events). In Java, one of the simplest ways in which a developer can enable support for this type of events is the use of AspectJ capabilities. [maybe more emphasis on AspectJ here] A developer can define multiple *pointcuts* and *advices* which inject small snippets that let the *TML* know that the events have occurred (Figure 2 demonstrates a typical aspect).



Figure 2 - the beep method is caught and the TML is notified

After developers have “wired” their code to the TML, it’s time to define usability experiments. Each experiment is a set of usability scenarios the developer wishes to track. For simplicity we denote usability scenarios as *Tasks.* The process of defining tasks in the TML is a very easy and is quite a dynamic process – one should simply write a ConGolog script which defines the task and let TML compile it (Figure 3 shows a simple ConGolog script which defines a basic usability scenario for a bank web-site).



Figure 3 – simple ConGolog script, which can later be compiled into a TML Task

Developers can later aggregate a couple of TML tasks into experiments, which can be activated for surveillance. TML records every activity and variable value changes that occur during its activation excluding those that are not part of any task which belongs to one of the active experiments. Any activity record is comprised by the following 4-touple: (name, timestamp, pre-condition status, tasks set[[1]](#footnote-1)), while variable records comprised of a 3-touple: (name, new value, timestamp).

At any point, developers can issue an analysis of the recorded data, against any task/experiment. TML offers a few basic analytics [ this is still a work in progress, currently only a basic set ] , but also supports exporting of the data into a CSV format for other analysis.

Supported GonGolog syntax [Assaf]

Currently the Compiler supports the following syntactic expressions:

|  |  |
| --- | --- |
| **Syntax** | **Semantics** |
| Precondition-axioms(<label>) = <boolean-expression>; | Pre-condition definition |
| Label <label> : {<label-body>} | Label definition. The label body can comprise of recursive label expressions. Sequential expressions depict a chronological order of events. |
| <activity>; | Activity representation. Any activity can only be written inside a Label container. |
| [<container-body>] | Container for multiple choice operators |
| | | Or operator. Specifies a selection between multiple choices. Can be aggregated to accommodate multiple “or” statement. |
| <bool-variable> ? <label/activity> | Stop and wait for the variable to become true[boolean expression will be supported in the future]. |

Tasks Analysis [Eli]

The library provides default analyzer for the recorded data.

Analyzing can be done on data related to specific task, or entire experiment.

Currently, the analysis compares the structure of the task to the recorded data. Thus, the analyzer checks whether the recorded data is consistent with the task’s activities, in the same order of their appearance in the task, and makes sure that all pre-conditions of each activity were met.

If either of the activities doesn’t appear, or the order is incorrect, or one of the pre-conditions is not met, the task is considered to be not fulfilled.

The experiment is considered to be fulfilled only if all tasks were fulfilled.

1. The set of tasks which relates to the specified activity [↑](#footnote-ref-1)