Quality of reusable game software: Empowering developers with automated quality checks

Wim van der Vegt
Open University of the Netherlands)
Heerlen, The Netherlands
wim.vandervegt@ou.nl

Wim Westera
Open University of the Netherlands
Heerlen, The Netherlands
ORCID: 0000-0003-2389-3107

***Abstract*— This study presents a quality assurance methodology geared to reusable (serious) game software that is posted on the Gamecomponents.eu marketplace portal. This portal provides an online hub for the exchange of serious game software components. The quality assurance methodology comes with a running prototype in C#. It is based on a flexible, plug-in architecture, which allows to flexibly add new checks. The tool starts at solution/project file level rather than code level and covers a variety of issues, e.g. naming conventions, warnings, documentation, portability, namespaces and classes. Specific coding level tests check for portability of game software components. Software testing results show that the approach uncovers a large numbers of hidden flaws and issues that require to be fixed. The tool will empower developers to enhance the quality of their software during development and will contribute to the overall quality level of exposed game software at the Gamecomponents.eu portal.**

Keywords—game, component, software quality, plug-in, reuse

I. INTRODUCTION

Today's online platform economy [1] has disruptively altered the exchange and trade of physical and virtual goods. Online marketplaces connect sellers and consumers throughout the world and readily provide the associated financial and logistics services. In the domain of software development various online marketplaces have emerged to promote the reuse of software, both for hobbyists and professionals. In particular, video-game oriented marketplaces have managed to attract a lot of attention among game studios, independent developers, researchers and students. Most dominantly, these game development portals are driven by commercial game engine vendors (e.g. Unity, CryTec, Unreal), allowing their customer-base to expose and sell their engine-compliant plug-ins or graphic models. Software offered on these sites will only run on the specific game engine and cannot be reused elsewhere. In contrast, the recently launched Gamecomponents.eu portal offers platform-independence, as it is not bound to a specific game development platform. Whilst the other portals mainly focus on leisure games, Gamecomponents.eu explicitly addresses serious games (games with a serious purpose in e.g., in education and training). The portal has been funded by the European Commission to expressly support and amplify the serious gaming sector, which is recognized for its growth potential and its capability to address a wide range of societal issues, e.g. in education, health, citizenship, immigration, inclusion and media literacy. However, seizing the opportunities is hampered by the inherent fragmentation of the serious gaming landscape (small companies, limited harmonization, limited access to knowledge and advanced software). The Gamecomponents.eu portal is currently populated with some 50 open source, reusable game software components, covering a diverse range of pedagogically-oriented functionalities including learning analytics, game difficulty adaptation, affective computing, essay grading and many more. In order to be able to effectively deploy these components across a wide diversity of game engines, programming languages and target platforms, the RAGE Client-Side Asset Architecture (RCSAA) [2,3] was developed, which effectively decouples the components' code from both game code and operating system. The RCSAA is a lightweight component-based software architecture that greatly simplifies component integration in different technical environments.

To be able to attract a growing number of users to the portal, Gamecomponents.eu aims to develop into a high quality label. In contrast, existing gaming software portals display - given the diversity of suppliers - an inherent lack of software quality assurance: platform owners wave any responsibility for the suppliers' products, leaving users without a clue. Gamecomponents.eu has been seeking a methodology to support quality assurance of third-party game software that is posted on its site. This paper proposes a methodology for this, the RCSAA Quality Assurance methodology (RQA methodology), and presents a running prototype, the RCSAA Quality Assurance tool (RQA tool), which demonstrates its functioning. So far, the study is confined to software in C#, which is dominantly used as a programming language for games [2]. The RQA methodology uses and combines various approaches to cover and check a relevant subset of software features and metrics, including some key features of the RCSAA. First, the RCSAA is briefly explained. Second, starting points of the RQA methodology are specified. Third, the technical design of the prototype tool is presented, including preliminary proofs of its functioning. The study is concluded with a brief discussion of findings.

II. THE RCSAA

A. Brief summary of the architecture

The RCSAA [2,3] was created to support the portability and reuse of game software components across the diversity of programming languages, game development environments and target hardware and operating systems that are being used in practice. Being based on the component-based software engineering paradigm it provides a component model for creating reusable plug-and-play components. To remove incompatibilities as much as possible, the RCSAA relies on a limited set of well-established software patterns and coding practices aimed at decoupling abstraction from its implementation. This decoupling facilitates reusability of a component across different software systems with minimal

integration effort. It uses a Bridge software pattern [4], which is platform-dependent code implementing one or more interfaces, for allowing a component to call methods from the game engine without the need to have knowledge about the game´s implementation details. The RCSSA implementation also defines a base class for components and their settings and has a lightweight component manager that allows components to locate each other. Interfaces are used to define the available functionality of the bridge. Implementing these interfaces has to be done by the component user, which is the game developer, and is kept simple to implement and re-usable.

A detailed description of the RCSAA and its classes and operations can be found in [2]. The RCSAA has been technical and ecologically validated across multiple programming languages, development environments, and target delivery platforms.

B. RCSAA-compliant components in C#

Software in C# takes mostly place in Microsoft Visual Studio, a powerful IDE for programmers. In Visual Studio coding a software component involves 3 layers.

- Solution: The solution is the top layer that binds together one or more projects.
- Project: A project combines source code, references to other (external) code/libraries and a .Net framework choice (amongst other settings) into an output product.
- Output product: For C#, three different output product types are available:
    - Console application (text mode programs),
    - Graphics applications,
    - Libraries called assemblies.

RCSAA components are designed to be compiled into assemblies (dynamic link libraries), so they can be referenced to by serious games being developed. However assemblies are used for more purposes like unit test suites, so there is no 1:1 relation between assembly and RCSAA components.

The RCSAA maintains portability by sharing the source code amongst 2 or 3 projects. One project (mostly the one compiled against .Net 3.5 is the main project) and the other two use file linkage to compile exactly the same code against two other .Net framework versions: Portable .Net framework and its recent successor the .Net Core framework, respectively (in due time, the former is expected to be replaced by its successor, the .Net Core framework). By this combination any portability issues between these frameworks will show up as failure to compile one or more of the projects. The setup also confines component development to the common part between these .Net versions and limits the use of code interfering directly with the underlying operating system or game code. It enables usage in a wide range of game development environments, e.g. Unity3D [5], Xamarin [6], Xenko [7] and Desktop.

III. RQA METHODOLOGY

A. Findings from manual code review

A manual code review of selected RCSAA-compliant components was carried out for a series of 20 components to assess coding quality and RCSAA compliance. This review yielded a list of common RCSAA related pitfalls and other coding issues that usually aren't covered by regular automated unit testing (used to test small sections of code, supply input and assert if the output is matching expectations). The issues identified in the code reviews [8] covered the following categories:

- Solution
- Project
- Coding issues
- Portability
- Demo project issues
- Unit tests

These include issues in e.g. projects' set-up, naming issues, use of classes from namespaces that will prevent the same code to be used in other .Net versions, the use of hardcoded paths to the developer's machine, files missing from (git) source control, incorrect or missing code documentation and other things. This should not necessarily disqualify the developers. An example of how 'easy' is it to make a mistake is the use of some classes in C# from the File.IO namespace to access the file system. The classes in this namespace are only supported in some particular branch of the .Net frameworks tree. It is however lacking in the Portable .Net and .Net Code frameworks that are most commonly used by Xamarin for mobile phone apps and UWP apps. With the proposed project setup, any attempts to use these classes would produce a failure to compile the non .Net 3.5 projects.

Also, various specific issues with respect to the RCSAA showed up. Since the RCSAA-compliant components should be portable across a wide range of development platforms and deployable to a broad range of (mobile) target platforms, they should not rely on external libraries that interfere with portability. In C# it means that the usage of assemblies or namespaces should be checked against a list of ones known to cause portability issues (e.g. File.IO namespace of which multiple classes only exist in a limited set of .Net versions, all targeting desktop environments only).

B. Methodological frame

Given the reusability aim of the Gamecomponents.eu marketplace and the RCSAA, the RQA methodology should preferably match the following requirements:

- Automation: To avoid laborious manual effort, the RQA methodology should be suited for creating an automated tool.
- Target users: Game software developers (suppliers) should be able to access this RQA tool and use it for assessing their software quality during component development, while automating detection of the greater part of the code reviews' findings. This means that the developers remain in control and the tool is not used to provide an authorized rating or approval stamp for components.
- Reporting: The output of the analysis is an advisory report that developers use in a formative way to improve the components quality and architectural uniformity, leading to improved integration and better acceptance.

- Scope: The methodology is not meant to validate correctness of core models, algorithms, or other component payloads, but instead should produce more generic quality metrics and RCSAA-compliancy tests. The latter include the discovery of RCSAA components, a check of the Bridge implementation, the use of external libraries, and some more.
- Extensibility: System design should support the flexible inclusion of new quality checks.

IV. Technical considerations

A. Technical starting points

As indicated before, we so far restrict ourselves to software in C# and the Visual Studio IDE. Since the RCSAA has been demonstrated to be portable across different programming languages C#, TypeScript, Javascript, Java and C++), there are no principal barriers to extending the tool to other code bases later on.

In the tool, only assemblies and unit tests will be included. Applications (executables) that can be started from a command-line prompt are currently ignored because it's purpose, like demo project, calibration software, connection tester, is unknown.

Although the approach should not go into source code analysis, as this involves complex parsing to build a representation, we can still use the compiled output files to determine whether or not an assembly is an RCSAA-compliant software component by using either reflection or exploiting the excellent quality of de-compilation in C# [9, 10]. By design, the RCSAA exposes a base API that we can use to programmatically handle components once compiled. The actual component payload (i.e. the pedagogical functionality it implements), can in C# be treated as a black box, but still allows one to examine certain implementation details with reflection without further knowledge. The detection of the public API of a component would be an example.

B. Toward plug-ins

As it is good practice to keep software extensible and modular, use of a plug-in based architecture with one or more plug-ins for each type of file (solution, project, output) is a sensible approach. Plug-ins are compiled pieces of code that expose a common interface so that an application can load and use them without knowing the exact implementation details. In a plug-in based architecture, each test to be performed is a separate plug-in: each plug-in can be used much like a test case in a unit test suite, viz. a small self-contained test that returns a clear answer: pass or fail. The Bridge pattern, which is used in the RCSAA to similarly shield implementation details cannot be used as an alternative here, since it lacks the capability of plug-ins to deal with multiple implementations of the same interface. Likewise, test suites should not be used as they are mostly compiled into single assemblies that are not directly executable. Besides, a unit test does not allow for supplying new tests without recompiling all code involved. Also passing parameters to test suites, like the solution file to be checked, is difficult as they are directly oriented at code level. Altogether, a plug-in architecture is most suited. Treating the plug-ins independently, that is, omitting (data) dependencies between plug-ins, allows for keeping the system flexible and extensible without the need to recompile and redistribute all code.

C. Dealing with C# solutions, projects and assemblies

Solutions and projects in C# are described using configuration files with fixed extensions: *.sln and *.csproj. Project output is either a console application, a windows application or an assembly. Both type of applications share an extension *.exe, while for assemblies the extension is *.dll (Dynamic Link Library):

- Solution
  - Project #1
    - Compiled Output #1
  - Project #2
    - Compiled Output #2
  - Project #3
    - Compiled Output #3

Optionally, one could define a git repository, e.g. Distributed Version Control System (DVCS), as a top level transcending the solution level. This would enable checks on these repositories as well and make it easier to perform checks outside the developers working environment.

The hierarchical nature of the solutions, projects and outputs allows to do checks starting at any level and if applicable include deeper levels into the check. A simple check at solution level would extract for example all projects and schedule all applicable plug-ins for this project. Likewise a project level plug-in could schedule all applicable plug-ins for the projects output.

More complex multi-level tests could scan a solution for projects and determine which projects have an RCSAA software component as output and subsequently do specific checks on all of these (for instance to check if a pair or triplet of RCSAA software components, compiled against the .Net 3.5, portable .Net or Net Core framework, is present).

D. Plug-in interface design

Given the strict naming conventions of configuration and output files of Visual Studio solutions (*.sln), projects (*csproj) and their compiled output (*.exe, *.dll), a plug-in could be passed the filename as parameter and use the file's extension as a first check of whether or not it can handle the file and perform a test on it. For using plug-ins as vehicles of quality checks, we define a simple, yet effective, interface with three methods:

- Supports(filename): If this method returns true for the filename that is passed to it, the RQA application can invoke the plug-in to perform the actual test to the file (Fig. 1). Supports() can either do a simple check on the extension of the filename or perform a more detailed analysis first based on the file content.
- Initialize(host): Prior to execution of the plug-in, the Initialize() method is invoked to setup any additional configuration data.
- Execute(filename): Based on results, Execute() can either return true or false, signalling check success or failure. The host object passed in the Initialize call is used during the execute call to emit results for

creation of a spreadsheet after all testing is done. The host object contains code common to all plug-ins and means to prevents duplicate code.

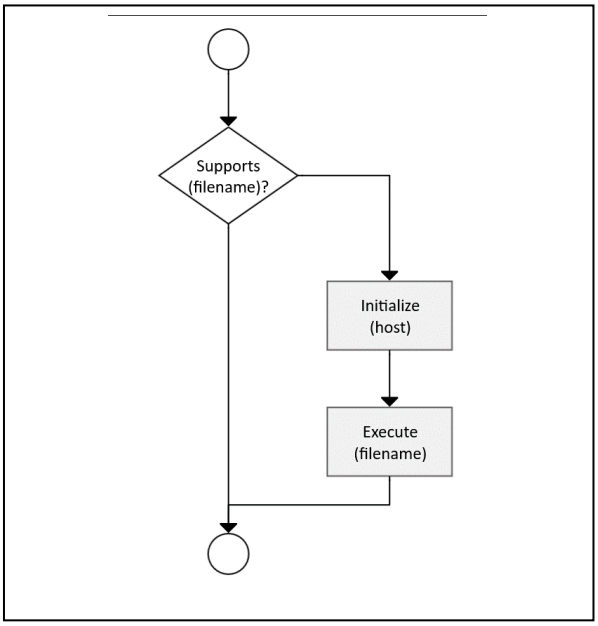

Fig. 1. *RQA plug-in method execution order*

E. Overview of plug-in tests

The RQA tool allows for implementing plug-ins at 4 different levels (Solution, Project, Assembly and Unit Test). Table 1 shows an overview of the plug-ins.

Two of the plug-ins, the Project File Reader and the Output detection plug-in invoker, respectively, are used to schedule other plug-ins at lower levels in the solution/project hierarchy.

Solution level checks in Table 1 are mainly about naming of components (Portable.Net and .Net Core components should have an 'Asset_Portable' and 'Asset_Core' suffix so they can easily be kept apart). The default namespace is expected to be 'AssetPackage' and for both the Portable.Net and .Net Core components a 'PORTABLE' symbol need to be defined which is used to fix some incompatibilities in the BaseAsset class. Because this check involves all projects in a solution it cannot be performed at an individual project level.

We use existing NuGet package creation code [11] to perform a number of checks (all necessary to build such package). These checks involve checking if all files are properly linked in the Portable.Net and .Net Core versions and if all metadata (e.g. author, copyright etc.) is present in all projects of the solution. The NuGet plugin produces a *.nuspec file, which is subsequently used to create the actual package containing the sources and all compiled component versions for easy deployment.

Project level checks also includes detection of any warnings left and documentation issues by building the project in such way that these warnings and messages are emitted. Finished code should not contain any warnings. If not solvable and/or intentional they can be suppressed in the source code. Documentation issues should also be fixed. Documentation is usually a remaining category that often is incomplete. This Xml documentation is used in Visual Studio for IntelliSense popups when using components.

TABLE I. OVERVIEW OF PLUG-INS IN THE RQA TOOL.

Plug-in	Target	Description
NuGet Package Generation Check	Solution	This plug-in tries to create a NuGet package with an RCSAA component and its sources from a solution (and checks for necessary metadata).
Component Pair Detection	Solution	This plug-in reports the project types detected in a solution and checks the naming convention (suffix).
Project File Reader	Solution	This plug-in schedules plug-ins for all projects detected in a solution.
Conditional Compilation Symbol Check	Project	This plug-in checks if the PORTABLE symbol is set correctly for the Portable and .Net core Assemblies.
Xml Documentation Check	Project	This plug-in compiles a project and checks for Xml Documentation Issues.
Project Builder	Project	This plug-in compiles a project with Detailed Summary Output option (but without Xml Documentation Issues)
Code Analysis	Project	This plug-in compiles a project with Code Analysis option.
Output detection plug-in invoker	Project	This plug-in examines a project file and schedules plug-ins for its output (if an Assembly).
Code Complexity Analysis	Assembly	This plug-in decompiles an Assembly and reports McCabe's Cyclomatic Complexity Metric.
Dead Code Detection	Assembly	This plug-in decompiles an Assembly and reports Dead Code.
Detection of blacklisted classes	Assembly	This plug-in decompiles an Assembly and reports any blacklisted classed used.
Interface Detection	Assembly	This plug-in decompiles an Assembly and reports RAGE Interfaces used.

Coding level checks are about both RCSAA compliancy and more generic coding issues. The RCSAA needs the component class being derived from its Base class. More generic is the calculation of metrics such as McCabe's Cyclomatic Complexity [12] metric, which provides a measure for the amount of effort needed to fully test a method.

Portability level checks examine the use of namespaces and classes preventing re-use in other .Net frameworks. Between .Net 3.5 and the Portable and .Net code frameworks some major differences exists. File and Stream I/O in the more modern frameworks are completely asynchronous. The existing synchronous File and Stream API's found in the System.IO namespace are no longer present. Another problematic class is System.Diagnostics.Debug where the methods for diagnostic output reside. Besides differences in method availability and method signatures between .net frameworks, using this class might also not match with the game developer's choice for handling diagnostic output. RCSAA components should use the Log method of their base class instead. This method uses the ILog interface that can be implemented by game developers to output these messages how they want it.

Demo projects are not tested except for compilation issues. For instance, these projects often contain hardcoded paths to filenames on the developer's machine, preventing compilation in different environments.

F. Controlling the plug-ins

Plug-ins have originally been described as software patterns that are only minimally addressed as pluggable adaptors in [4]. Ever since, plug-ins have evolved into the base of software patterns such as Inversion of Control (IoC) or Dependency Injection (DI) [13]. These two patterns, however, focus on late binding of classes. Their principal goal is to be able to configure an application by using some of the available plug-ins to supply implementations for functionality instead of hard wiring them at compile time. The RQA tool has a different purpose: to use all available plug-ins for performing check on RCSAA components. DI literature mentions a Service Locator as an alternative concept that is used to get decoupled access to all plug-ins [13], but some consider it an anti-pattern [14].

Recently, however, both IoC and DI [4,13] have made their way into the common toolkits of programmers. Both IoC and DI allows for defining plug-in containers that can contain multiple plug-ins, each of which define a particular interface, that allow for retrieving one or more of these plug-ins implementing the interface. This allows for composing plug-in applications by externally defining which plug-ins are inserted into the container. This mechanism effectively reverses the control from the application to the external configuration file.

Examples of IoC containers for .Net are Unity [15] (not to be mistaken for Unity3D, a game development IDE), Castle Windsor [16], Autofac [17]. The literature also pointedly mentions the Managed Extensibility Framework (MEF) [19-21], which is not considered a full IoC/DI framework (it implements a subset of IoC functionality), but focuses on providing more generic plug-in functionality to applications instead of application configuration functionality only. For the RQA tool MEF was selected as most appropriate as it is a) an integral part of .Net 4, b) focusses solely on the plug-in problem, c) there is no official specification of IoC, and d) quite some IoC/DI features are not relevant for RQA tool purposes (such external configuration of plug-in containers and runtime application configuration).

In C# we can chose to compile plug-ins for MEF into separate assemblies or group them to keep the number of assemblies low. MEF is used to search those assemblies and find all plug-in classes that implement the desired interface.

G. Scheduling

In order have a common place for often used functionality such scheduling other plug-ins for discovered projects or assemblies, the Initialize() method is passed a Host class instance, implementing this commonly needed functionality. The Initialize() and Execute() methods are not called immediately but scheduled for later execution.

When a plug-in is active and scans for other suitable plug-ins for a deeper level in the solution/project/assembly hierarchy, it places these plug-ins into a queue together with its filename parameter. For example, a solution level plug-in can identify appropriate plug-ins for each subordinate project it discovers. By scheduling new plug-ins for later execution, it is secured that only one plug-in runs at a time. The RQA tool also keeps track of already executed checks (i.e. plug-in/filename parameter pairs) to prevent duplicate checks. To this end, it keeps a separate list of executed plug-ins and their parameters. This effectively prevents unwanted recursion behavior.

In order to cold start, RQA tool only needs to schedule all plug-ins for the solution or project to be examined. This way, the RQA tool keeps taking the front item of the queue and execute it until the queue is empty. During this process the queue may grow and shrink as a result of new plug-ins being executed inserted in the queue, and jobs being completed and removed. This approach also makes it easy to generate documentation in a logical order.

V. Results

The RQA tool including its 14 plugins (cf. Table 1) was tested on four different solutions. Three solutions contained RCSAA components selected from the gamecomponents.eu portal and one was an empty solution. The RCSAA components were designed to offer game storage functionality, facial emotion detection, and automated difficulty adaptation, respectively. The output reported by the RQA tool includes general diagnostic comments, specific things to reconsider such as the size and nature of the public API and warnings, e.g. about incorrect documentation and compilation issues.

A. The empty solution

The empty solution was handled correctly and resulted in a small number of failures as output, caused by no projects being detected.

B. The client-side game storage component

This component offers component developers as well as game developers a centralized function to manage and store tree-like datasets (e.g. player IDs, player scores, domain models) that should be shared with other components, without the need to bother about RCSAA-compliancy and the usage in different game engines. The RQA tool correctly discovered a portable assembly version, a demo project and a unit test. It also reported 43 documentation issues, 4 of which are in the main component class and the remaining 39 in underlying data storage classes. A public API as large as 56 methods was reported. This large number may be attributed to some internal classes made public for unit testing. For 5 of the methods a high cyclomatic complexity of 10 or higher was reported. The RQA tool further reported 5 user warnings (using a #warning compiler directive) that served as reminders or to-do items. Finally, RQA detected the presence of 5 bridge interfaces, 2 of which were part of the RCSAA and 3 newly defined in this component.

C. The Real-time Facial Emotion Detection Component

This component uses artificial emotional intelligence to unobtrusively cover unbiased facial expressions of emotion from any image, either from a still, a video file, a video stream or a webcam. It classifies the emotion in real time and returns a string value for this, representing either happiness, sadness, surprise, fear, disgust, anger or the neutral face [22]. The emotion detection component is one of the few client-side component at gamecomponent.eu that is not available as a portable or .Net core assembly. This is caused by a C++ wrapper dll used to gain access to Dlib [23] methods capable of detecting faces and subsequently calculating 68 facial landmarks, which the component uses to classify the facial emotions. The RQA tool correctly identified a failure to compile a portable/.Net core project. No documentation issues where detected. A public API of 24 methods was detected. Two out of 20 methods had a cyclomatic complexity of 10 or higher. The RQA tool detected 3 user warnings and 1 compiler

warning in the component. Finally it revealed the use of the blacklisted Path class from the System.IO namespace, which will result in portability issues.

D. The Adaptation and Assessment component (TwoA)

The Adaptation and Assessment component (TwoA) provides a fuzzy-logic based algorithm for the real-time adaptation of task difficulty to user skill. This component assumes that there are multiple tasks in the game of varying difficulty levels, which ideally can be controlled parametrically. It expects a player performance metric as input and also uses time on task as an indicator. Based on the history of player performance it updates the player's expertise rating and returns the optimal difficulty level for the next task to be assigned. Through continued re-iteration of task difficulty and player's expertise level, it guides the player along the optimal learning curve. A detailed description of the adaptation mechanism is given in [24]. The RQA tool reported that a .Net core project was missing. A public API of 158 methods was reported, which seems rather large for a component that might do its work with only a few. Five methods of 186 analyzed ones where reported having a cyclomatic complexity of 10 or higher. Most of the API methods where related to accessing settings of the underlying model. The use of a single bridge interface was reported. No documentation issues or warnings for this component were detected.

VI. Discussion

The main purpose of the test cases reported above has not been in revising the tested software components as such, but in validating the plug-in architecture of the RQA tool, currently featuring 14 plug-ins, and its capability of appropriately detecting and reporting a wide range of issues. The tests with the existing RCSAA components uncovered a large number of hidden flaws and issues, many of which require to be fixed. Even though these components had been subjected before to an extensive manual code review, be it at an earlier stage, many issues still surfaced. The RQA tool shows to be able to detect a variety of useful issues, particular with respect to documentation. Correct code documentation is important as Visual Studio's IntelliSense depends on it and documentation gaps are among the first things a component user would encounter. The RQA tool also confronts the developers with overly complex methods that might need refactoring into smaller, easier to test, methods. Also being confronted with the public API seems useful: keeping the size of the API down by information hiding is a well-known principle of object oriented programming. Issue detection in solution and project layout resulted mainly in reports on missing .Net core projects.

Some of the plugins need further investigation. There is a slight mismatch in the number of public methods reported in the public API and in the cyclomatic complexity calculations. Properties and field to the public API report could be easily added as the information needed is provided once the assembly is decompiled. Unit test execution would be much more laborious as the complete Assert class needs to be remapped from a non-distributable Visual Studio assembly to an RQA supplied version. Determining the correct replacement method is the main issue. But once completed it will open possibilities to execute unit tests in various local settings, viz. altering the decimal separator character or date time format, and inject soft errors into the bridge code to test the components ability to cope with it. It would also enable measuring code coverage to assess unit test quality.

The use of both ICSharpCode.Decompiler and Mono.Cecil might need some consolidation to prevent duplicate method signature generation code. Dead code detection results in too many false positives to be useful. Also, it should be noted that the RQA tool was tested only with single RCSAA component solutions; component suites including multiple components are not supported in the current RQA version. Support for modern Distributed Version Control Systems (DVCS) like git would add a new top-level starting point. It would allow checks for DVCS and solution/project file mismatches. Ongoing work is extending the list of blacklisted classes that interfere with portability to other .Net frameworks.

So far, usability of the RQA tool has not been at the core of the study. Now that the system prototype has been tested, usability factors should be taken care of, including an accompanying guide on how to interpret the report and how to solve reported issues. Even in its current prototypical state, the RQA tool already has proven to provide valuable quality information on RCSAA components in a single report, potentially saving a lot of manual checking. The tool will empower developers to enhance the quality of their software during development and will contribute to the overall quality level of game software, in particular software exposed at the Gamecomponents.eu portal.

Acknowledgment

This work has been partially funded by the European Commission's H2020 project RAGE (Realising an Applied Gaming Eco-System); http://www.rageproject.eu/; Grant agreement No 644187.

References

[1] D. Farrell, F. Greig, F. and A. Hamoudi, The Online Platform Economy in 2018. New York: JP Morgan Chase & Co Institute, 2018. Retrieved from https://www.jpmorganchase.com/corporate/institute/document/institute-ope-2018.pdf

[2] W. Van der Vegt, E. Nyamsuren and W. Westera, "RAGE Reusable Game Software Components and Their Integration into Serious Game Engines", in: Proceedings of the 15th International Conference on Software Reuse (ICSR 2016), Bridging with Social-Awareness, G.M. Kapitsaki & E. Santana de Almeida (Eds.). Basel: Springer International Publishing, 2016, pp. 165-180.

[3] G.W. Van der Vegt, W. Westera, E. Nyamsuren. A. Georgiev and I. Martinez Ortiz, "RAGE architecture for reusable serious gaming technology components", International Journal of Computer Games Technology, Article ID 5680526, 2016. DOI: 10.1155/2016/5680526

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design patterns: elements of reusable object-oriented software. London: Pearson Education, 1994.

[5] Unity3D, product website, 2019. Retrieved February 27, 2019 from https://unity3d.com/

[6] Microsoft, Xamarin Documentation, 2019. Retrieved March 5, 2019 from https://docs.microsoft.com/en-us/xamarin/

[7] Xenko, Open-source C# Game Engine, 2019. Retrieved March 5, 2019 from https://xenko.com

[8] W. Westera, K, Stefanov, W. Van der Vegt, E. Nyamsuren, K. Bahreini, E. Kluijfhout, P. Moreno Ger, M. Freire, A. Georgiev, A. Grigorov, P. Boytchev, D. Griffiths, B. Fernández Magnón, S. Mascarenhas, I. Martinez Ortiz and M. Hemmje, RAGE Deliverable 1.1 – Applied gaming asset methodology. Retrieved March 5, 2019 from https://research.ou.nl/admin/editor/dk/atira/pure/api/shared/model/researchoutput/editor/bookanthologyeditor.xhtml?id=1036792

[9] JetBrains, dotPeek .Net decompiler, 2019. Retrieved March 5, 2019 from https://www.jetbrains.com/decompiler/

[10] Telerik, Telerik JustDecompile, 2019. Retrieved March 5, 2019 from https://www.telerik.com/products/decompiler.aspx

[11] Microsoft, Creating NuGet packages, 2017. Retrieved March 5, 2019 from https://docs.microsoft.com/en-us/nuget/create-packages/creating-a-package

[12] T.J. McCabe, "A complexity measure", IEEE Transactions on Software Engineering vol. 2, issue 4, , pp. 308-320, July 1976.

[13] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern", personal blog, 2004. Retrieved March 5, 2019 from https://www.martinfowler.com/articles/injection.html

[14] M. Seemann, "Service Locator is an Anti-Pattern", personal blog, 2010. Retrieved March 5, 2019 from http://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/

[15] Unity container project, Welcome to Unity Container Documentation, Github, 2019. Retrieved March 5, 2019 from https://unitycontainer.github.io/

[16] E. Mays, "Getting Started with Dependency Injection Using Castle Windsor", Codementor community, 2017. Retrieved March 5, 2019 from https://www.codementor.io/copperstarconsulting/getting-started-with-dependency-injection-using-castle-windsor-4meqzcsvh

[17] Autofac Project, Inversion of Control Container, 2013. Retrieved March 5, 2019 from https://autofac.org/

[18] M. Seemann, Dependency Injection in .NET. Shelter Island, NY: Manning Puvlications, 2011

[19] Microsoft Archive, Mef, Github, 2019. Retrieved March 5, 2019 from https://github.com/MicrosoftArchive/mef

[20] Microsoft, Managed Extensibility Framework (MEF), 2017. Retrieved March 5, 2019 from https://docs.microsoft.com/en-us/dotnet/framework/mef/

[21] S. Hanselman, MEF - Managed Extensibility Framework with Glenn Block, Hanselminutes Podcast, 2009. Retrieved March 5, 2019 from https://www.hanselminutes.com/148/mef-managed-extensibility-framework-with-glenn-block

[22] K. Bahreini, W. Van der Vegt and W. Westera, "A fuzzy logic approach to reliable real-time recognition of facial emotions", Multimedia Tools and Applications, online version, Feb 6, pp. 1-14, 2019. doi=10.1007/s11042-019-7250-z

[23] Dlib, C++ Library, 2018. Retrieved March 5, 2019 from http://dlib.net/

[24] E. Nyamsuren, W. Van der Vegt and W. Westera, "Automated Adaptation and Assessment in Serious Games: a Portable Tool for Supporting Learning". In: Proceedings of the Fifteenth International Conference on Advances in Computer Games 2017 (ACG2017). Lecture Notes in Computer Science, vol 10664, M. Winands, H.J. van den Herik and W. Kosters (eds.). Springer, Cham, 201-212. DOI:10.1007/978-3-319-71649-7_17