View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Open University of the Netherlands Research Portal

Portability of Serious Game Software Components

Citation for published version (APA):

van der Vegt, W., Westera, W., Kurvers, H., & Nyamsuren, E. (2019). Portability of Serious Game Software Components. In *IEEE Conference on Games 2019: London, United Kingdom 20-23 August 2019* (pp. 221-228). IEEE. https://doi.org/10.1109/CIG.2019.8848094

DOI:
10.1109/CIG.2019.8848094

Document status and date:
Published: 26/09/2019

Document Version:
Peer reviewed version

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

https://www.ou.nl/taverne-agreement

Take down policy
If you believe that this document breaches copyright please contact us at:

pure-support@ou.nl

providing details and we will investigate your claim.

Downloaded from https://research.ou.nl/ on date: 17 Jan. 2020

Open Universiteit
www.ou.nl

Portability of Serious Game Software Components

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

Hub Kurvers
Open University of the Netherlands
Heerlen, The Netherlands
hub.kurvers@ou.nl

Enkhbold Nyamsuren
Open University of the Netherlands
Heerlen, The Netherlands
e.nyamsuren@gmail.com

***Abstract*— In recent studies, a component-based software engineering framework (RCSAA) has been proposed to accommodate the reuse of game software components across diverse game engines, platforms, and programming languages. This study follows up on this by a more detailed investigation of the portability of a RCSAA-compliant game software component across three principal programming languages: C#, JavaScript (TypeScript), and Java, respectively, and their integration in game engines for these languages. One operational RCSAA-compliant component in C# is taken as the starting point for porting to the other languages. For each port, a detailed analysis of language-specific features is carried out to examine and preserve the equivalence of transcompiled code. Also, implementation patterns of required RSCAA constructs are analysed for each programming language and practical workaround solutions are proposed. This study demonstrates that the software patterns and design solutions used in the RCSAA are easily portable across programming languages based on very different programming paradigms. It thereby establishes the practicability of the RSCAA architecture and the associated integration of RCSAA-compliant game components under real-world conditions.**

Keywords— Serious game, applied game, reuse, component, asset; gamification; portability; RAGE

I. INTRODUCTION

While the leisure game market is being dominated by a handful of global players (e.g. Sony, Nintendo, Microsoft) supporting their propriety game consoles and thus establishing de facto industrial standards, the serious gaming market is scattered over a large number of small independent players, all using different programming languages, game engines and platforms [1]. As a result of this the cross-platform use and reuse of software is not possible. Despite the fact that lively vendor-bound developer communities and marketplaces have emerged, e.g. linked with Unity, CryEnginegame, Cocos2d, or Unreal game engines, the exchange and reuse of software is limited and bound to the respective platforms. The portability of game software between different game platforms dramatically fails. As a consequence, serious games lack the generality and harmonisation that would be required for their wider distribution and usage in a diversity of operational conditions. This hampers the (partial) reuse of existing game software in new games, and unnecessarily increases production costs and time-to-market [2].

To establish and preserve the portability of game software across the wide diversity of game engines, software systems and programming languages, the RAGE client-side asset architecture – RCSAA [3] has been proposed. The RCSAA is a generic component-based software engineering framework [4,5] that accommodates the reuse of software components across different parent environments. Proofs of concept of this component-based architecture have been provided that demonstrate its compliance with the following basic requirements: 1) minimal dependencies on external software frameworks and 2) interoperability between components, 3) portability of components across both development environments and target platforms and 4) portability of components across different programming languages. To this end, dummy implementations ("Hello World") were established in C#, Typescript, Java and C++, respectively [3]. In a subsequent paper the technical integration of a selected RCSAA-compliant software component in C# into a running example game in the MonoGame engine was analysed and reported, as well as integrations with the Unity game engine and Xamarin [6]. Current paper follows up on these studies by providing a more in depth and systematic investigation of portability by focusing on selected RSCAA-compliant software components rather than on dummy components.

In this study we start off from an existing RCSAA-compliant software component that is available in C#, and then investigate the implications of porting it from C# to TypeScript/JavaScript and Java, respectively. TypeScript is used as a superset of JavaScript that adds static typing, which can be used by Integrated Development Environments and compilers to check for coding errors. By examining and comparing these three different code bases, we have covered the predominant languages used in game development [7], namely compiled languages (C# for desktop and mobile games, Java for server-based systems) and interpreted languages (HTML5/JavaScript for browser games), respectively.

Below we will first provide a recap of the rationale and principles behind the RCSAA and the set of communication modes it supports. Next, for each of the language ports (from C# to TypeScript/JavaScript and Java, respectively) we will identify and analyse language-specific features that may affect the equivalence of transcompiled code, and examine the implementation of required RSCAA constructs. As a final check, each ported software component will be integrated and tested in a game engine based on the respective programming language.

II. THE RAGE CLIENT-SIDE ARCHITECTURE (RCSAA)

A. Components

The purpose of the RCSAA architecture [3,6] is to enable developers to easily include extra functionalities, viz. through portable software components, in their game development projects. The RCSAA defines a component model for creating a reusable plug-and-play component. Its client-side focus refers to the fact that the components need to be locally integrated into the parent system, which is one of many game

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

engines. The RCSAA serves to minimise incompatibilities of the components with these engine.

The component model conforms to common norms of Component-Based Development [4,5]: 1) a component is an independent and replaceable part of a system that fulfils a distinct function; 2) a component provides information hiding and acts as a black box; 3) a component communicates strictly through a predefined set of interfaces that guard its implementation details.

An RCSAA-compliant component may either be a source code file or a compiled program file. Components are enriched with machine-readable metadata, such as keyword classifiers, descriptions, and information about versions, licenses, component dependencies and programming language used. In accordance with the general definition of an "asset" by the W3C ADMS Working Group [8] the components may also include additional artefacts that are not to be compiled and run as software, but provide additional guidance and support such as tutorials, manuals, licenses, configuration tools, authoring tools and other resources. The full component with all artefacts included can be packaged for distribution. Examples of RCSAA-compliant components are available on the gamecomponents.eu marketplace portal, which is financially supported by the Horizon 2020 Programme of the European Commission. This portal constitutes a platform-independent technology transfer hub that allows suppliers and users of game software components to connect. Currently, over 40 components have been developed and exposed on the portal, which cover a wide range of functionalities particularly tuned to the pedagogy of serious gaming, e.g. player data analytics, real-time emotion recognition, real-time arousal detection, rule-based adaptation, game difficulty balancing, procedural animations, virtual characters, essay grading, sentiment analysis, interactive storytelling, social gamification and many other functions [9].

B. The RCSAA design solution

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. The parent system is supposedly a game engine, but all considerations are applicable to other software systems as well.

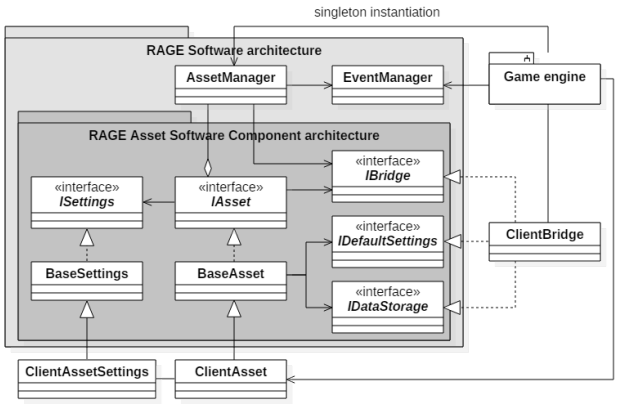

Fig. 1. Class diagram reflecting the internal structure of an RCSAA-compliant game component.

Figure 1 shows the UML class diagram of the RCSAA Component. Here, the IAsset class, which is defined as an interface, provides the abstract definition of the component including the fields, properties and methods required for its operations and communications. The BaseAsset class implements the set of basic functionalities of the component following the definitions provided by IAsset. IBridge provides a standardised interface that allows the component to communicate with external technologies such as the game engine or a remote service. The ISettings interface ensures in accordance with the abstract definition in the IAsset interface that every component has the basic infrastructure for managing a unique component ID, type, settings, version information, etc., which is then realised by the BaseSettings class.

The following design solutions are used in the architecture (A detailed description of the RCSAA and its classes and operations can be found in [3]):

- No interference with the user interface
  To avoid platform-dependent code, the component only provides processing functionality by returning processed data to the game engine (e.g. calculating user performance metrics based on logged behaviours). The component operates under the hood and thus preserves the creative freedom of game designers and developers to control the graphics, the user interface and the look and feel of their game.

- Coordinating agent (Asset Manager)
  Since various components may be linked together to express aggregates, a coordinating agent is needed: the Asset Manager, which is implemented using a Singleton software pattern [10]. It handles registration of components and exposes methods to query these registrations.

- Bridge pattern
  For allowing a component to invoke game engine code, the Bridge software pattern [10] is used, which is platform-dependent code implementing one or more interfaces. As such, the components are not aware of the actual implementation details. These implementations can be re-used by multiple RCSAA components. Alternatively, the communications could use the Publish/Subscribe pattern [10,11,12] through the Event Manager, which is initialised by the Asset Manager during its Singleton instantiation.

- Settings
  The component offers basic capabilities of storing configuration data (settings), be it delegated through the Bridge to the game engine. Storage also includes localisation data (string translation tables), version information and dependency information (dependency on other components' versions).

- Programming language's features
  Components largely rely on the programming language's primitives, standard features and libraries to maximise the compatibility across game engines supporting that language. Therefore, components should delegate the implementation of required operating system features to the actual game engine using the Bridge, for example, for the actual storage of runtime data.

These design solutions are used to cover all important communication modes, while maintaining component uniformity and keeping the game developer in control of the component integration and use. Furthermore, the design solutions make RCSAA-compliant components very well suited for unit testing [13,14] and working with stubs or mock objects [15], as the component is uninformed about the details of the parent environment. The available communication modes have been described in more detail elsewhere [6] and include: (a) component to component; (b) component to game engine; (c) component to web-service; (d) game engine to component and (e) message broadcasting.

III. Real-world implementations of the RCSAA

Among many RSCAA-compliant components currently available in the gamecomponents.eu portfolio, we will use the TwoA (Adaptation + Assessment) component [16], which uses a fuzzy-logic based algorithm for the real-time adaptation of task difficulty to user skill. The TwoA component assumes that there are multiple tasks 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 [16]. The C# implementation of the TwoA component has been extensively described elsewhere [6]. The C# version will be used as a reference for analysing and discussing the details of language conversions to JavaScript/TypeScript and Java, respectively. The TwoA component consists of 2572 lines of C# code in 12 classes. In both conversions the actual C# source code was used as the starting point and converted to JavaScript/TypeScript and Java on a line by line base. This method was chosen as a large part of the code, e.g. method bodies, have identical syntax in all three languages and the method highlights any remaining conversion issues.

A. Conversion from C# to TypeScript/JavaScript

1) General considerations about JavaScript

While the object-oriented nature of C# makes it relatively straightforward to implement all features of the RCSAA presented above, the JavaScript implementation is more complex. JavaScript is a prototype-based programming language, which is not ideal for reusability. It has several drawbacks concerning programming convenience, code maintenance, refactoring and more importantly quality control. For example, there is no native support for common object-oriented encapsulation structures such as classes and namespaces [17], which not only hinders direct translation of architectural elements but also reduces the readability of the code. Furthermore, in JavaScript, there is no compile-time type checking, which can result in severe errors during reuse of the architecture by developers. This set of errors tends to surface at run-time and not at compile time as is the case with programming languages that support type checking.

TABLE I. Comparison of constructs in C#, TypeScript and JavaScript.

Programming language		
C#	***TypeScript***	***JavaScript***

// a namespace
namespace AssetPackage {
 	
// an interface
public interface IAsset {}
 	
// a class
public class BaseAsset : IAsset {
 	
// a constructor
public BaseAsset() {}
 	
// a method
public Boolean LoadSettings
(String filename) {
return true;
}
 	
// a property
public IBridge Bridge {
get; set;
}
}
}
	// a namespace
	module AssetPackage {
	
	// an interface
	export interface IAsset {}
	
	// a class
	export class BaseAsset implements IAsset
	{
	
	// a constructor
	constructor() {}
	
	// a method
	public LoadSettings
	(filename: string): boolean {
	return true;
	}
	}
	}
		// a namespace
		var AssetPackage;
		(function (AssetPackage) {
		
		// a class
		var BaseAsset = (function () {
		
		// a constructor
		function BaseAsset() {}
		
		// a method
		BaseAsset.prototype.LoadSettings =
		function(filename) {
		return true;
		};
		return BaseAsset;
		}());
		AssetPackage.BaseAsset = BaseAsset;
		})(AssetPackage||(AssetPackage = {}));

1) Using TypeScript as an intermediate

To avoid these problems, we first ported the component code to TypeScript, which is a superset of JavaScript and can be automatically transcompiled into JavaScript. TypeScript supports common object-oriented patterns without compromising inherent advantages of JavaScript such as flexibility and cross-platform support. It enables encapsulations based on classes, interfaces, and modules (analogous to namespaces in C#). Other features supported by TypeScript are type definition, type inference, and compile-time type checking. Therefore, patterns in C# can be translated

almost one-to-one to patterns in TypeScript. Table I compares language constructs in C#, TypeScript, JavaScript, respectively.

The C#-based implementation of the RCSAA can be easily migrated to TypeScript-based implementation. The RCSAA deals only with code that implements logic and avoids code for the user interface, thereby reducing errors. For example, a recent study [18] suggests that 80% of the errors in JavaScript programs are related to the Document Object Model (DOM). The RCSAA implemented in TypeScript does not allow components to have direct access to the parent system (e.g., a web browser). Therefore, components cannot interact directly with the DOM, eliminating the 80% portion of DOM-related errors by design. In addition, one-third of the remaining 20% of the errors is type related and can be minimized by a stricter type checking offered by TypeScript [19].

A confusing difference between C# and TypeScript is that the type and variable names are swapped and the location of the method return types is different in TypeScript. Additionally, it is recommended to avoid concepts such as 'var' (inferred types) in C# and the type 'any' in TypeScript, not only to improve the compiler's type checking, but also to improve code readability and self-documentation.

2) Transcompiling Typescript into JavaScript

As TypeScript, with its object-oriented syntax and type checks, only exists at compile time, most object orientation is lost in the transcompiled code. JavaScript interpreters are highly optimised to obtain acceptable performance and omit e.g. type checking: when calling transcompiled TypeScript code from JavaScript at runtime, no type checking occurs. Besides adding additional checks on method input parameters passed to component methods, API documentation may be an even more important way to prevent these issues of passing parameters.

Although the JavaScript code in Table I is structurally very similar to the TypeScript code, it is clear that all type checking is omitted in the transcompiled code. Furthermore, the transcompiler generates some additional code to mimic the object-oriented concepts with pure JavaScript code. Finally, code comments are also present in the generated JavaScript.

a) Run time checks

Some checks that can be easily performed in C# are not possible in JavaScript, for instance checking at runtime whether or not a particular interface is implemented. Such check is not possible as the required information, called Run-Time Type Information (RTTI), does not exist in the resulting JavaScript. As a workaround, the BaseAsset.getInterface method parameter was changed into a method name parameter instead of an interface type and internally uses the bridge's prototype to check for the interface method's presence. As this does not check for the complete interface it is not as strongly typed as the C# code.

b) Variable declaration and scope

JavaScript and TypeScript also display different behaviours with respect to variable declaration and scope. In compiled languages, variable declaration and scope follow the location in the code. Firstly, a variable cannot be used in compiled languages before it is declared and compiled, and secondly, initial assignments are performed at the location of the declaration. In JavaScript, variable declarations are silently moved to the top of the code block, but the initial assignment of a value is not moved [20]. This easily results in the presence of uninitialised variables or masking of a global variable with an uninitialized variable, if a local variable happens to have the same name as a variable in an outer scope. This kind of bugs can be very hard to trace.

c) Data formats

Data is also stored differently as the component architecture does not prescribe a particular data format. Instead, the RCSAA focuses on the best natively supported format. For C#, only XML is supported natively in the .NET version 3.5 and above. Although .NET 3.5 introduces some JSON support with a *DataContractJsonSerializer* class [21], this class is not supported by the .NET 2.0 version used by Unity3D. For JavaScript, JSON is currently the only format natively supported in recent browsers, which all implement a built-in JSON object [22].

Beside the data format, there is also an important difference in behaviour when (de)serializing data. Unlike C# and Java, it is not possible to fully restore a class instance in JavaScript (and thus in TypeScript), including class methods. Restoring JSON creates an object with only data and thus results in a complete loss of all class methods. For this reason, the *BaseSettings* class in RCSAA cannot contain any methods and de-serialization code is located in the *BaseAsset* class.

d) Other issues in JavaScript

Other differences in behaviour originate from the interpreter nature and single-threadedness of JavaScript. The use of methods such as setTimeout (for broadcasting messages) imply that these messages are sent when the interpreter is idle, so when all other executed code has finished. As a result, it is possible that subscribers receive updates if they have subscribed after the publication of the update. In C# and Java, doing the same will not have any effect as the messages are sent immediately, thus before any further subscriptions could take place.

Finally, the interpreted JavaScript easily allows for self-modifying code which is hard to achieve in compiled languages such as C++, C# or Java. This feature, although powerful, was avoided as it cannot be ported easily to any other compiled language.

B. Porting to Java

1) General considerations

Java is an object-oriented language that has a long history, predating C#. In contrast to C#, which has been in continuous development and has been extended with new language features, Java has known a long period of minor development, exposing only few new features in the past years. Although Java has many different features, it lacks some of the more modern features present in C#. Most obvious is the lack of clear syntax for properties as found in C#. In C#, the compiler takes care of converting a property into accessor methods and backing storage. Properties are often used in C# to expose public values that can be read or written by other code. Java, instead, mimics properties with a naming convention (get/set method name prefixes) and therefore forces a programmer to re-implement trivial implementation code repeatedly, with an increased chance of coding errors. Nevertheless, not all methods that have a get or set prefix mimic a property. A 3rd party project called Project Lombok [23] addresses this

omission and enriches Java with a compact property syntax that compiles automatically into high-quality code.

In C#, all data types, even numbers and Boolean values, are treated as objects. In C#, statements such as *7.ToString()* or *(9+1).ToString()* are perfectly legal as the numeric values are treated as objects. The C# compiler optimises this code during compilation. In contrast, Java has C/C++ alike non-object primitive types [24] such as *int*, *double*, *boolean* and object counterparts such as *Integer*, *Double*, and *Boolean* that box their primitive counterpart types. Primitive types are not objects and do not have methods. The mix of primitive types and objects forces a bad practice [25] of a manual optimization by the programmer instead of delegating the optimization to the compiler.

The code in Table II looks very similar as both C# and Java have extensive support for object-oriented principles. The definition of a property in Java shows how it relies on more extensive coding and naming conventions to emulate the very compact C# property syntax.

2) Specific Java porting issues encountered

a) The Asset Manager

The first step in porting the C# code for the TwoA component to Java was to update the Asset Manager code to be aligned with its C# counterpart. Updating was largely a matter of refactoring classes, method and field names and adding, as is a common practice in Java, the 'final' keyword to most fields and method parameters, marking them immutable.

TABLE II. COMPARISON OF THE CONSTRUCTS IN C# AND JAVA.

Programming language	
C#	***Java***

// a namespace
namespace AssetPackage {
 	
// an interface
public interface IAsset {}
 	
// a class
public class BaseAsset : IAsset {
 	
// a constructor
public BaseAsset() {}
 	
// a method
public Boolean LoadSettings
(String filename) {
return true;
}
 	
// a property
public IBridge Bridge {
get; set;
}
}
}
	// a package
	package
	eu.rageproject.asset.manager;
	
	// an interface
	public interface IAsset {}
	
	// a class
	public class BaseAsset extends
	IAsset {
	
	// a constructor
	public BaseAsset() {}
	
	// a method
	public Boolean LoadSettings
	(final String filename) {
	return true;
	}
	
	// a property backing field
	private IBridge bridge;
	
	// getter
	public IBridge getBridge() {
	return this.bridge;
	}
	
	// setter
	public void setBridge(final
	IBridge bridge) {
	this.bridge = bridge;
	}
	}

In the *AssetManager* test suite, we encountered issues with the test suite implementation of *IDataStorage* interface on the *Bridge* class and in particular the usage of the '*user.dir'* environment property.

Despite its description 'User working directory' it points to the directory where the Java Virtual Machine (JVM) was started. In our case, this turned out to be the Visual Studio Code installation directory, which is write protected. Swapping the environment property for 'user.home', which is the user home directory and is writable, solves the issue.

The *ILog* interface resulted in some issues with the *LogLevel* enumeration. Whilst in C# we could simply define the *LogLevel* values by combining *Severity* enumeration values using a logical OR operator, in Java, it is needed to define these values as separate *EnumSet* fields and create the *LogLevel* enumeration based on *EnumSet* fields. Although the *EnumSet* fields alone might seem sufficient, they would not provide type-safety of the *LogLevel* enumeration.

The *AssetManager* singleton patterns were re-implemented using a single value enum instead of a plain class. According to Bloch [25], this is by far the simplest yet best solution as the single instance is thread-safe, enforced by the compiler and has no issues with deserialization.

b) Unsigned integers

During porting the TwoA core functionality, issues arose with the *SimpleRNG* class as it used unsigned integers, which are not present in Java [16]. The common solution is to use bigger signed integers (so 32-bit unsigned integers are stored in 64-bit signed integers) [27]. As the C# *SimpleRNG* code used logical bit shift operators, it needed to be rewritten and tested separately in order to yield the same results. Additionally, we needed to add some suffixes to some numeric values (like L for long) in order to aid the compiler.

c) Date

The *Date* API suffers major flaws and is heavily deprecated [28]. We recoded all C# *DateTime* using the new Java 8 *LocalDateTime* class. Use of this newer class results however in a new issue. Both *ZonedDateTime* and *LocalDateTime* lack a parameter-less constructor, which make them incompatible with JAXB XML deserialization. A solution is to register JAXB *XmlAdapters* to enable a custom conversion between text and both *ZoneDateTime* and *LocalDateTime*.

d) XML

The XML formatted logging showed issues with floating point to string conversions as used by the *String.format()* method. It uses the decimal separator defined by the operating system which is not necessarily the dot required for XML. Adding a *Locale.ROOT* to the *String.format()* method fixed these conversions [29]. XML serialization also suffers from differences between the Java and C# naming conventions resulting in a mismatch in character case of XML tags. Adding *XmlElement* annotations to all affected methods specifying the correct case solved this issue.

e) Other issues in Java

In Java, the 'const' keyword is reserved, but not implemented [30]. The closest alternative is the 'final' keyword that is used to mark fields and method parameters as immutable once given a value, which makes it more equivalent to the C# 'readonly' keyword. However, when

used on classes, 'final' is closer to the C# 'sealed' keyword that prevents a class to be sub-classed. Finally, when applied to methods the 'final' keyword prevents overriding the method.

Java also lacks support for static constructors. However, it allows for one or more static code blocks in a class, so one can simply move the static constructor's code of the *Cfg* class into such block.

Exceptions in Java have some fundamental differences with those in C#. They need to be either caught on the spot or be specified as a *throws* annotation in the method signature. The latter feature leads to accumulation of these annotations up in the class hierarchy and to an exception bubble upwards in this hierarchy until they are actually caught with a try/catch block. In C#, no such annotations are necessary, and exception handling remains a responsibility of the programmer.

The resulting Java version of the TwoA component has 2503 lines of code in 13 classes.

IV. Integrating RCSAA ports in different game engines

Table III presents the overview of component integrations carried out across different programming languages and game engines. The C# version of the TwoA component was integrated and used in an exemplary target game, called TileZero [6]. This game is a derivative of the popular turn-based board game Qwirkle (http://www.mindware.com). It focuses on problem-solving for developing spatial, mathematical, and fluid reasoning skills [31] TileZero allows for parametrically generating a variety of problems of various difficulty levels, it can greatly benefit from the TwoA component, which returns after each task completion the recommended difficulty level of the subsequent task, thereby optimising the player's learning curve. TileZero was implemented in the MonoGame game engine, which is a portable open-source Mono-based and OpenGL-based game engine (monogame.net; [32]).

TABLE III. Integration cases of TwoA ports.

Case		
Language	***Game***	***Engine***
C#	TileZero	MonoGame
C#	I/O simulation	Unity
C#	I/O simulation	Xamarin
JavaScript	I/O simulation	Cocos2D-JS
Java	I/O simulation	Emergo

Tests of the integration with other C# game engines, viz. Unity and Xamarin, have been reported elsewhere [6]. All of these game engines support a large number of leading target user platforms, covering different hardware configurations and operating systems including Windows desktop, iOS, Android and Windows Phone. RCSAA-compliant ports of TwoA from C# to JavaScript (TypeScript) and Java were integrated in Cocos2D-JS game engine (https://cocos2d-x.org) and the Emergo game engine [33], respectively. For testing in these engines game engine code was used that simulates the interactions of TileZero between the component and the game engine. This way laborious ports of the full TileZero game to both Typescript and Java could be avoided.

A. Integrating the C# component in MonoGame

The integration of the C# version of the TwoA component in the TileZero game has been described in detail in [6]. Basically, the component uses its Asset Manager to make its instance accessible for the game. The Bridge pattern is used to enable the TwoA component to call methods from the game and the MonoGame engine without the need to have knowledge about the game´s implementation details. The Bridge can also realise additional interfaces that enable a component to delegate common functionalities to standard libraries provided by the game engine. For instance, the component may request the game engine to load or save files. By delegating such generic functionalities from the component to the game engine and the RCSAA libraries, simplifies component development and leaves more time for the developer to spend on the implementation of the core gamification and pedagogical functionalities. Details of integrating C# versions of TwoA in Unity and Xamarin are also in [6].

B. Integrating the TypeScript/JavaScript port in Cocos2D-JS

The TypeScript version of the TwoA component was integrated in the Cocos2D-JS engine. This engine relies on JavaScript and HTML5. After creating a project in Visual Studio that included a Cocos2D-JS library written in JavaScript and the TwoA component written in TypeScript, the project could be transcompiled and integrated into a single JavaScript file simulating gameplays of TileZero.

In JavaScript, the Asset Manager and the Bridge work similar to the C# version. From the perspective of the component, it does not matter in what form or where the settings file is stored, as these details are abstracted from the component by the Bridge pattern and implemented by the game engine, thus proving full control to the game developer. Cocos2D-JS uses Local Storage [34,35] to manage the component's settings file. Local Storage was introduced in HTML5 for web applications to store data locally within the user's browser. Local Storage is distinct from cookies and, among many differences, allows storage of several megabytes of data. Ideally, Local Storage is persistent, and data can remain across sessions as well as after closing the browser. However, this persistence also depends on the browser's history and privacy settings and therefore should be used cautiously. Hence, having a persistent storage on the user's platform may remain problematic for components written in JavaScript/TypeScript.

Another notable difference from the C#-based implementation is storage of the component's settings in JSON format rather than XML format. JavaScript provides a native interface for parsing a JSON string into objects. In the TwoA case, the settings for scenarios and players are automatically de-serialized into JavaScript objects by using the available method for this.

Finally, the Bridge object enables logging. Note that the TwoA component provides information to be logged, but as is the case for settings, it does not specify where and in what format the logged information should be managed. The TwoA component only knows that logging functionality is available via a Log method inherited from the BaseAsset class. The Log method locates an ILog interface subsequently uses it. The

Bridge's ILog implementation calls the logging functionality of Cocos2D-JS, which then returns a given string to the browser's console.

C. Integrating the Java port in the Emergo game engine

Emergo [33] is a server-based environment written in Java that allows educators to create scenario-based serious games for students using the separate modules, module configuration and scenario building functionality that Emergo supplies. Educators and students access this environment using a web browser. In order to integrate the TwoA component into the Emergo environment, it needs to be wrapped inside an Emergo module so that the TwoA component can be instantiated, becomes available within the Emergo toolkit and its API be exposed to the scenario building functionality of Emergo. The wrapping module supplies the game engine code for the Bridge object that the TwoA component expects and uses to save and load its data.

Because of its server-based architecture, the Emergo engine needs to track and update game data for multiple users. However, the TwoA component currently does not support multiple users updating their scenario ratings. As a result, the bridge implementation must cope with this and save the scenario and gameplay data as files on a per user base. This is quite possible, because the Asset Manager supports registration of multiple instances of a single RCSAA component and returns unique ids for each component registration. This allows the bridge, which is needed to save and load scenario and gameplay data, to be attached to the individual instances of a component instead of being attached to the Asset Manager. With the addition of a user-id field to the bridge it is easy to keep the user data separate from the data of other users. As Emergo is relying on a database for main storage, a more sophisticated integration solution is to have the bridge save scenario and gameplay data as a blob in a database table instead of using the file system. It demonstrates the benefit of the RCSAA component being agnostic to where and how the bridge code actually stores data.

Storing the scenario and gameplay data as individual records is also possible but harder to implement as it would require parsing the XML-formatted data generated by the TwoA component and determining which data is changed. Likewise, results of a query would have to be converted into the expected XML format. Such approach would, however, conflict with the principles of the RCSAA as it requires the bridge implementation to have detailed knowledge of the data format being requested to be stored or retrieved by the component. It would also require knowledge of the file identifiers used by the TwoA component for the scenario and gameplay data as their data format differs.

D. Validation of the ported versions

The functioning of the ported component versions was tested by using the C# implementations as a reference. For TwoA, extensive validations of the C# version of the algorithm integrated in the TileZero game have been reported elsewhere, both using empirical performance data [16] and using machine-against-machine simulations [6]. A similar range of testing matches could then be simulated for the TypeScript/JavaScript and Java implementations by sampling from the performance data (match duration and outcome) produced by the C# simulation. The correct functioning of the ported TwoA algorithms could be confirmed by the identical learning curves that were found. This is quite straightforward as the core code uses the same model. However, the significance of these tests is not in verifying the correct implementation of the core functionalities of the components as such, but more importantly, in providing real-world proofs of the portability and practicability of RCSAA-compliant components: the relevant RCSAA-code constructs can be ported across the three examined programming languages without principal issues.

V. Discussion and Conclusion

In this study, we have provided further, practical evidence for the (ecological) validity of the RCSAA [3] as a framework for use and reuse of game software in different technical environments. Conclusions can be summarised as follows. First, it was shown that component development and reuse of components are simplified by delegating generic functionalities to the game engine and to the RCSAA. Second, the RCSAA simplifies porting of the components across game engines supporting a common programming language. Third, the RCSAA also simplifies translation of the component's implementation to other programming languages. The power of the RCSAA is not limited to the potential reuse of components, but is also based on the efficient reuse of existing libraries, either from the RCSAA or from the game engine in use. To maximise the reusability of components among different games, the components do not directly link with the game´s user interface and exchange only the basic information with the game engine. In the TwoA component, for example, the core code of the component responsible for difficulty adaptation requires only the exchange of string IDs and a few numerical values such as the duration of a task. This qualifies the integration of RCSAA components as "lightweight", which may promote its adoption.

In the case of JavaScript, being very different from object-oriented languages, we have demonstrated that conversion of the architecture's implementation from one language to another is simplified considerably by using TypeScript as a transient language. TypeScript was deliberately designed to be similar in structure to object-oriented languages and C# in particular. This makes a translation of C# code into TypeScript code a relatively simple operation. The Bridge pattern employed by the RCSAA adequately shields components from platform-specific implementations of necessary functionalities. As a result, components are easily portable and reusable across different platforms. The benefits are particularly obvious for the JavaScript-based implementation. Browser specific issues are often a plague for web applications based on JavaScript. The Bridge pattern effectively decouples components from browser-specific implementations thereby minimizing browser compatibility problems and increasing their reuse potential. Furthermore, the architecture encourages developers to concentrate all browser-specific functionalities at bridges, thereby increasing readability and decreasing effort needed for refactoring. Game developers can address cross-browser portability with minimal prior knowledge of the components. Having this convenience is especially important for JavaScript-based applications that are notorious for being difficult to refactor.

The Java version shows that the RCSAA is flexible enough to even run in a complex server-side system such as Emergo and that the Bridge interface can easily be mapped onto an SQL database system as well, extending its portability even further.

In summary, we have demonstrated how the RCSAA promotes reusability at two distinct levels. First, the architecture promotes plug-and-play reusability of a software component among different game engines. Second, software patterns and design solutions employed by the architecture are reused in different programming languages including JavaScript, which does not natively support object-oriented design. Reusability at both levels is essential for the successful adoption of software components in the domain of serious games, which is notably suffering from a variety of platforms, game engines, and programming languages. Given some issues that surfaced with integration, be it minor ones, a cautious and prolonged investigation is needed of the practical factors and conditions that might corrupt seamless component integration, both for C# and other languages.

ACKNOWLEDGMENT

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

REFERENCES

[1] J. Stewart, L. Bleumers, J. Van Looy, I. Mariën, A. All, et al., The Potential of Digital Games for Empowerment and Social Inclusion of Groups at Risk of Social and Economic Exclusion: Evidence and Opportunity for Policy. Centeno, C. (Ed.), Brussel: Joint Research Centre, European Commission, 2013.

[2] S.J. Warren and G. Jones, "Overcoming Educational Game Development Costs with Lateral Innovation: Chalk House, The Door, and Broken Window. The Journal of Applied Instructional Design 4 (1), 51-63, 2014.

[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] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, et al. , Technical concepts of component-based software engineering, Volume II. Carnegie Mellon University, Pittsburgh: Software Engineering Institute, 2000.

[5] S. Mahmood, R. Lai and Y.S. Kim, "Survey of component-based software development". IET software 1 (2), 57-66, 2007.

[6] 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). Basel: Springer International Publishing, 2016, pp. 165-180,.

[7] G.L. Saveski, W. Westera, L. Yuan, P. Hollins, B. Fernández Manjón, et al., "What serious game studios want from ICT research: identifying developers' needs". In: Proceedings of the Games and Learning Alliance conference (GALA 2015), Basel: Springer International Publishing, 2015, pp. 32–41. DOI:10.1007/978-3-319-40216-1

[8] M. Dekkers. Asset Description Metadata Schema (ADMS). W3C Working Group, 2013. Retrieved February 27, 2019 from http://www.w3.org/TR/vocab-adms/

[9] W. Westera, W., Van der Vegt, K., Bahreini, M. Dascalu, et al., "Software Components for Serious Game Development". In T. Connolly & L. Boyle (Eds.), Proceedings of the 10th European Conference on Games Based Learning, October 6-7 2016, Paisley, Scotland: Reading UK,: ACPI, 2016, pp. 765-772

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design patterns: elements of reusable object-oriented software. London: Pearson Education, 1994, pp. 171-183.

[11] K. Birman and T. Joseph. "Exploiting virtual synchrony in distributed systems", Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87), Austin: 1987, pp. 123–138.

[12] P.T. Eugster, P.A. Felber, R. Guerraoui and A.M. Kermarrec, "The many faces of publish/subscribe", ACM Computing Surveys (CSUR) 35 (2), 114-131, 2003.

[13] E.J. Weyuker, "Testing component-based software: a cautionary tale", IEEE SOFTWARE 15(5), 54-59, 1998.

[14] S. Mahmood, R. Lai, Y.S. Kim, J.H. Kim, S.C. Park, et al. , "A survey of component based system quality assurance and assessment", Information and Software Technology 47(10), 693-707, 2005.

[15] F. Fowler, "Mocks Aren't Stubs", 2007. Retrieved February 27, 2019 from https://martinfowler.com/articles/mocksArentStubs.html

[16] 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. Cham: Springer2017, pp. 201-212. DOI:10.1007/978-3-319-71649-7_17

[17] A. Osmani, Learning JavaScript Design Patterns, Sebastopol, CA: O'Reilly, 2014.

[18] F.S. Ocariza Jr. and K. Pattabirama, A Study of Causes and Consequences of Client-Side JavaScript Bugs, IEEE Transactions on Software Engineering 43(2), 128-144, 2017.

[19] Microsoft, TypeScript Language Specification, 2016. Retrieved February 27, 2019 from https://github.com/Microsoft/TypeScript/tree/master/doc/TypeScript%20Language%20Specification.pdf

[20] Rauschmayer, A., JavaScript variable scoping and its pitfalls, 2011. Retrieved February 27, 2019 from http://2ality.com/2011/02/javascript-variable-scoping-and-its.html

[21] Microsoft, System.Runtime.Serialization.Json Namespace, 2018. Retrieved February 27, 2019 from https://msdn.microsoft.com/en-us/library/system.runtime.serialization.json(v=vs.90).aspx

[22] W3schools, JavaScript JSON, 2018. Retrieved February 27, 2019 from https://www.w3schools.com/js/js_json.asp

[23] Project Lombok. 2018. Retrieved February 27, 2019 from https://projectlombok.org

[24] Oracle, Primitive Data Types (The Java™ Tutorials > Learning the Java Language > Language Basics). Oracle, 2017. Retrieved February 27, 2019 from https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

[25] J. Bloch, Effective Java, New Jersey: Addison-Wesley, 2008.

[26] Project Nayuki, Unsigned int considered harmful for Java, 2018. Retrieved February 27, 2019 from https://www.nayuki.io/page/unsigned-int-considered-harmful-for-java

[27] N. Coffey, What is the Java equivalent of unsigned?, 2008. Retrieved February 27, 2019 from https://javamex.com/java_equivalents/unsigned.shtml

[28] B. Evans and R. Warburton, Java SE 8 Date and Time, 2014. Retrieved February 27, 2019 from http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html

[29] Oracle, Class Locale, 2017. Retrieved February 27, 2019 from https://docs.oracle.com/javase/7/docs/api/java/util/Locale.html#ROOT

[30] Oracle, Java Language Keywords, 2008. Retrieved February 27, 2019 from https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

[31] A.P. Mackey, S.S. Hill, S.I. Stone and S.A. Bunge, "Differential effects of reasoning and speed training in children", Developmental Science 14 (3), 582-590, 2011.

[32] J. Pavleas, J.K.W. Chang, K. Sung and R. Zhu, "Learn 2D Game Development with C#", New York: Apress, 2013, pp. 11-40.

[33] A. Slootmaker, H. Kurvers, H. Hummel and R. Koper, "Developing scenario-based serious games for complex cognitive skills acquisition: Design, development and evaluation of the EMERGO platform", Journal of Universal Computer Science 20(4), 561-582, 2014.

[34] M. Casario, P. Elst, C. Brown, N. Wormser and C. Hanquez, "HTML5 local storage", in: HTML5 Solutions: Essential Techniques for HTML5 Developers. New York: Apress, 2011, pp/ 281-303.

[35] W. West and S.M. Pulimood, "Analysis of privacy and security in HTML5 web storage", Journal of Computing Sciences in Colleges 27 (3), 80-87, 2012.