A Methodology on Extracting Reusable Software Candidate Components from Open Source Games

Apostolos Ampatzoglou
Department of Informatics
Aristotle University
Thessaloniki, Greece
apamp@csd.auth.gr

Antonios Gkortzis
Department of Information Technology
Alexander Technology Educational Institute
Thessaloniki, Greece
gkortz@it.teithe.gr

Ioannis Stamelos
Department of Informatics
Aristotle University
Thessaloniki, Greece
stamelos@csd.auth.gr

Ignatios Deligiannis
Department of Information Technology
Alexander Technology Educational Institute
Thessaloniki, Greece
ignatios@it.teithe.gr

ABSTRACT

Component-Based Software Engineering (CBSE) focuses on the development of reusable components in order to enable their reuse in more systems, rather than only to be used to the original ones for which they have been implemented in the first place (i.e. development for reuse) and the development of new systems with reusable components (i.e. development with reuse). This paper aims at introducing a methodology for the extraction of candidate reusable software components from open source games. The extracted components have been empirically evaluated through a case study. Additionally, the component candidates that have been extracted are available for reuse through a web service.

Categories and Subject Descriptors

D.2.13 [**Reusable Software**]: *Reusable Software – domain engineering, reusable libraries, reuse models*

General Terms

Measurement, Design and Experimentation

Keywords

Component selection, class dependencies, metrics, case study

1. INTRODUCTION

In most countries video games are a prevalent entertainment form, concerning their social and economic impact. Particularly, according to Consumer Electronics Association [4] reports, the worldwide revenue of the game industry increased from nearly $11 billion in 2003 to nearly $50 billion in 2007. In addition to that, according to the same data, playing games has outperformed many other entertainment forms, like listening to music and watching movies.

Additionally, according to [1 and 11] computer game development lifecycle is so intense that implementation phase is in need of techniques that will shorten the product time to market, while minimizing the effort spent for debugging and testing activities. In [6, 7 and 9] the authors propose innovative architectures that enhance the reusability of games and game engines. Such architectures produce more stable and extensible software, increase interoperability, improve robustness and scalability, minimize coupling between modules and shorten the architecture learning curve. Finally, in [5 and 12] reuse opportunities in game development are examined. However, the way of extracting software components that can be used in game development has not been considered in the game engineering literature. Games can be large, complex software projects and despite their individuality there are a lot of common concepts. Thus, techniques that can aid developers through reuse opportunities are worth exploring.

In software engineering, software components are typically equivalent to software packages and classes [13]. In [2] the authors suggest that software components can be extracted on the basis of pattern instances [8] and compared the internal quality of patterns with packages and classes. One drawback of the above mentioned approaches is the fact that the set of classes that are involved either in the package or the design pattern instances are not compileable, i.e. there are dependencies to classes out of the package or the pattern instance. In such cases, the reuser should modify the code of the package or the pattern instance in order to be adopted in the target system without compile errors. Further problems are the understanding of the game code (program comprehension) and the cost of component adaptation. Thus, the minimization of external dependencies of the candidate components is expected to reduce its adaptation cost.

In this paper, we propose a methodology for automatically identifying well-formed subsets of Java classes from open source games, further referenced as *Component Candidates*. Such sets of classes might not be ready to use as components for black box reuse, in the sense that they are not necessarily compilable and their functionality has not been examined. However, the extracted component candidates are considered fitting for white-box reuse purposes, since they are expected to exhibit minimum external dependencies, useful functionality and reusability. At this point it is necessary to clarify that the paper deals neither with subsystem identification nor with feature extraction, but only with the identification of candidate components, which should be further examined so as to discover their provided functionality and semantic meaning.

MindTREK'12, October 3 – 5, 2012, Tampere, Finland.

The main idea of the methodology is to perform a dependency analysis on a UML class diagram and extract component candidates based on the class structural dependencies and make them available for reuse by others. In order to evaluate the Candidate Components that are extracted with the proposed methodology, we applied it to fourteen (14) games, and the extracted components have been evaluated through a case study. The findings of the paper have been used in order to populate the dataset of a web repository on components for game development[1]. At this point, the web repository holds about 3 million component candidates, extracted from 135 open source java games and 16.015 java classes.

2. METHODOLOGY

The proposed methodology aims at extracting the most easily reusable sets of classes from a Java open source project, using path-based strong components algorithms. The methodology is applied to every class of the system and produces a set of candidate components. The prerequisite for applying the methodology is the creation of a graph that depicts the dependencies among classes of a system. An example of such a graph, created from an existing open source game, namely Scotland Yard[2], is presented in Figure 1. The outcome of the methodology will be the suggestion of sets of classes that are as independent as possible and provide significant functionality to the rest of the system.

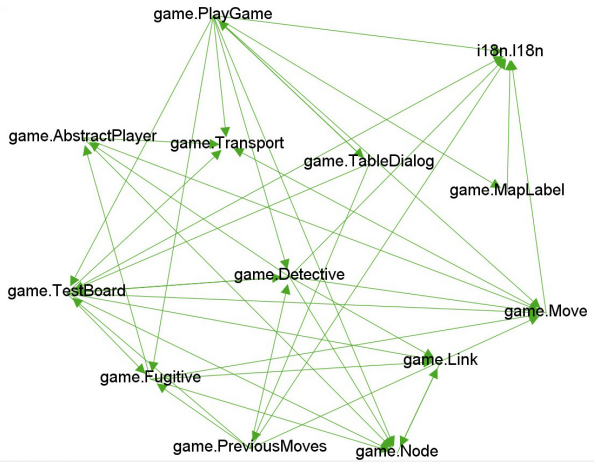

Figure 1. Dependency Graph of Scotland Yard OSS Game

Next, for every class the following steps are performed:

step 1. Create a data structure where *Candidate Components* are stored. The structure is a dynamical two dimensional array. The number of rows defines the maximum number of classes that can be included in a *Candidate Component*. The number of columns represents the count of possible *Candidate Components* that can be used for any component size. Create the first *Component Candidate*, of size 1, for one class of the system.

step 2. Identify the classes that the participants in the *Candidate Components* are connected to.

step 3. Place the dependencies in a list sorted by their number of external dependencies in a descending order.

step 4. For every dependency in the list create an updated *Component Candidate* and place it in the corresponding position in the data structures.

step 5. Return to step 2, for every *Component Candidate* created in the previous step, according to the order that they have been added in the data structure. The process stops if the maximum number of components is reached or if there are no external dependencies.

An example of how the methodology is applied is presented below. Suppose the system of Figure 2 and the dependency graph of Figure 3.

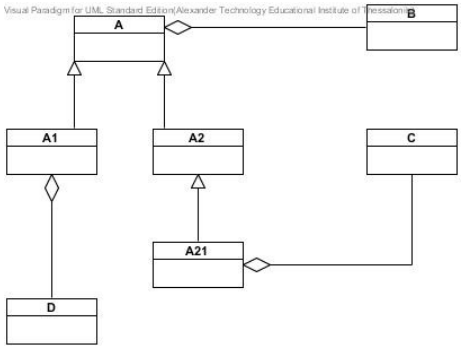

Figure 2. Methodology Demonstration

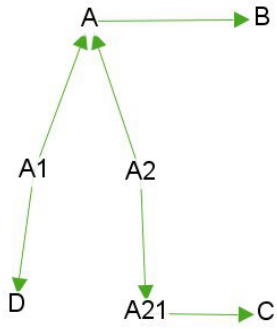

Figure 3. Methodology Demonstration Dependency Graph

Lets suppose that the starting class is A. The first step creates a *Candidate Component* of *size* 1 that contains only the starting class.

Table 1. Candidate Components (1st pass) – Starting Class A

Size 1	A
Size 2	

Since class A has only one dependency (class B), in second pass there will be only one *Candidate Component* with size 2, that consists of classes A and B.

Table 2. Candidate Components (2nd pass) – Starting Class A

Size 1	A
Size 2	A,B

[1] http://www.percerons.com, Patterns and Components Repository Extracted from Open Source Software.

[2] http://sourceforge.net/projects/scotland-yard/

Classes A and B do not have any other dependencies. Thus, the process of creating *Candidate Components* that start from class A, is completed.

In order to have a better understanding on the steps executed in every algorithm pass, we present an example on how algorithm works if starting class is A1. Similarly to the previous case, pass 1, creates only one *Component Candidate* that contains only the starting class, i.e. A1.

Table 3. Candidate Components (1st pass) – Starting Class A1

Size 1	A1	

Class A1 has two dependencies A and D. Using these two classes the algorithm will create two *Candidate Components* with size two. First *Candidate Component* A1,D will be created because class D has less external dependencies than class A.

Table 4. Candidate Components (2nd pass) – Starting Class A1

Size 1	A1	
Size 2	A1,D	A,A1

The candidate component extraction algorithm is recursive and traverses the dependency graph in a depth first manner. Thus, in 3rd pass the starting node will be A1,D. This *Candidate Component* has one external dependency, i.e. class A. Thus, the first size 3 component that will be created is A,A1,D.

Table 5. Candidate Components (3rd pass) – Starting Class A1

Size 1	A1	
Size 2	A1,D	A,A1
Size 3	A,A1,D	

Candidate Component A,A1,D has one external dependency, class B, and will create an additional Component Candidate of size 4 as shown in Table 6.

Table 6. Candidate Components (4th pass) – Starting Class A1

Size 1	A1	
Size 2	A1,D	A,A1
Size 3	A,A1,D	
Size 4	A,A1,B,D	

Candidate Component A,A1,B,D has no external dependencies and the algorithm will backtrack to a smaller size *Candidate Component* that has not been checked, i.e. components of size 2, A,A1. The corresponding component candidate has two external dependencies, classes B and D. The new *Candidate Components* of size 3 are A,A1,B and A,A1,D, with the former component not to be added in the list, since it already exists. Thus the list after pass 5 is presented in Table 7.

Table 7. Candidate Components (5th pass) – Starting Class A1

Size 1	A1	
Size 2	A1,D	A,A1
Size 3	A,A1,D	A,A1,B
Size 4	A,A1,B,D	

From Table 7 and Figure 3 we can observe that there is no distinct path beginning from class A1. Thus, the algorithm has identified six *Candidate Components* for a reuser that wants to reuse class A1, as shown in Table 8.

Table 8. Final Candidate Components – Starting Class A1

Component Candidate	External Dependencies	Component Size
A1	2	1
A1, A	2	2
A1, D	1	2
A1, A, D	1	3
A1, A, B	1	3
A1, A, B, D	0	4

After applying the method for all classes of Figure 2, the final set of Candidate Components is created and presented in Table 9.

Table 9. Final Candidate Components – All Classes

Size 1	A	A1	A2	A21	B	C	D
Size 2	A,B	A1,D	A,A1	A,A2	A2,A21	A21,C	A,C
Size 3	A,A1,D	A,A1,B	A,A2,B	A,A2,A21	A,A21,C	A2,A21,C	A,B,C
Size 4	A,A1,B,D	A,A2,A21,B	A,A2,A21,C	A,A21,B,C			
Size 5	A,A2,A21,B,C						

3. CASE STUDY

In this section we evaluate the components that are retrieved with the selected methodology. The validation method is a case study. Case studies are fitting evaluation methods when a large dataset is available and when the environment that the method is applied is not controlled. The case study organization is similar to [2] where the authors performed a case study to evaluate the reusability of design patterns.

3.1 Case Study Plan

The aim of this case study is to investigate the quality characteristics of the software components retrieved by applying the proposed methodology, against components that are retrieved based on software packages. The steps that have been followed during case study execution are the following:

- Define research questions
- Build the dataset
- Identify the method of comparison
- Execute case study
- Analyze and report the results

3.2 Define Research Questions

The research question of the paper can be described by the following scenario: "A developer wants to implement a specific requirement. He identifies a class that provides the main functionality that he wants to implement, through keyword search. Such keywords may be domain entity names (e.g. detective) and player actions (e.g. ShortestDistance). Which classes should be selected, modified and reused in the final project?" In our research we investigated three alternatives for the reuser:

- Select a component based on the proposed methodology [Alternative A1 – Candidate Component].
- Select the package that the class belongs to [Alternative A2 - Package].

- Select all packages to which any class of the proposed component of Alternative A1 belongs to [Alternative A3 - PackageSet].

3.3 Build the Dataset

In our case study we performed the proposed methodology on fourteen (14) open source games written in java. The projects have been randomly mined from an open-source repository and are therefore of different quality levels, in order for our results not to be affected by the quality of the subjects. After applying the methodology in all classes of the projects (2.803 classes), 577.319 component candidates have been proposed. In order to evaluate the component candidates we created a twelve column dataset, for each of the component candidates. The list of columns is described below:

- Size of Alternative A1
- External Dependencies of Alternative A1
- Functionality of Alternative A1
- Reusability of Alternative A1
- Size of Alternative A2
- External Dependencies of Alternative A2
- Functionality of Alternative A2
- Reusability of Alternative A2
- Size of Alternative A3
- External Dependencies of Alternative A3
- Functionality of Alternative A3
- Reusability of Alternative A3

3.4 Identify the Method Comparison

In order to compare the three alternatives, we selected several structural quality metrics, which have been used for assessing size, independency (Efferent Coupling), functionality (Afferent Coupling) and reusability of the components retrieved by each class selection alternative. The selected metrics are described in Table 10. We have preferred not to discuss the nature of the measurements in more detail. The interested user can access their definitions in the primary studies in which metrics have been defined. The three alternatives have been compared based on descriptive statistics, frequencies and charts.

Table 10. Metrics Used in Evaluation

Attribute	Metric	Definition
Size	Number of Classes (NOC)	Counts the number of classes that are involved in the component
Dependencies	Fan Out (FO)	Counts the number of classes outside the component that are essential for the component to compile [10]
Functionality	Weighted Fan In (WFI)	Ratio of the number of classes outside the components that use at least one class inside the component, to the total number of classes outside the component [10]
Reusability	Reusability (R)	Equation that calculates the ease for a set of classes to be transferred to another system. Uses size, coupling, cohesion and messaging metrics [3].

3.5 Results

In Table 11 we present the descriptive statistics for the case study sample. As it is observed, the fourteen games considered had an average size of about 340 classes. The average component size that was extracted with Alternative A1 is almost 20 classes that have an average reusability index of about 5.4, the proposed set of classes is used by about 34% of the remaining classes of the system and in order to compile they need an average of 18 classes. Alternative A2 created component candidates of about 68 classes that have an average reusability index near 3.1. The proposed set of classes is used by about 8% of the remaining classes and in order to compile they need an average of about 20 classes. Finally, concerning Alternative A3, the proposed set of classes (about 165 classes) have an average reusability of about 3.6, they are used by 24.8% of the remaining classes of the system and in order to compile they need about 19 classes.

Table 11. Descriptive Statistics

	N	min	max	mean	std. dev.
Project Size (NOC)	576803	13.000	590.000	340.520	179.880
Alternative A1 (NOC)	576803	1.000	40.000	20.310	11.190
Alternative A2 (NOC)	576803	1.000	323.000	68.300	99.654
Alternative A3 (NOC)	576803	1.000	576.000	166.540	127.843
Alternative A1 (R)	576803	-16.460	177.794	5.476	4.426
Alternative A2 (R)	576803	- 1.937	35.162	3.090	2.590
Alternative A3 (R)	576803	- 1.937	35.162	3.672	1.700
Alternative A1 (FO)	576803	0.000	119.000	18.370	20.667
Alternative A2 (FO)	576803	0.000	108.000	20.210	23.251
Alternative A3 (FO)	576803	0.000	250.000	19.040	25.586
Alternative A1 (WFI)	576802	0.000	1.000	0.376	0.172
Alternative A2 (WFI)	543514	0.000	0.500	0.083	0.100
Alternative A3 (WFI)	540218	0.000	0.857	0.248	0.195

The results of Table 11 provide an outline on the structural quality characteristics of the class selection alternatives. However, so as to investigate if the above results are uniform among all possible candidate component sizes, we had to split the dataset and perform further analysis.

In Figures 4 to 7 we present line charts that represent the average number of classes, fan out, weighted fan in and reusability, for the components created with the three component selection alternatives. The x-axis represents the size of components, whereas the y-axis of the graphs represents the value of the metric score.

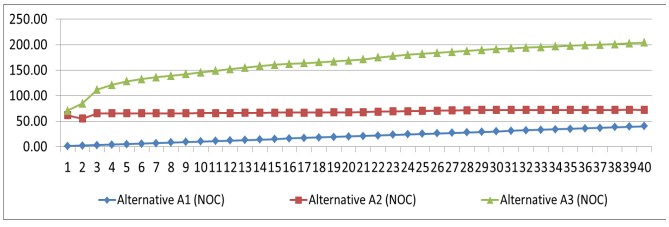

Figure 4. Size of Component Candidate

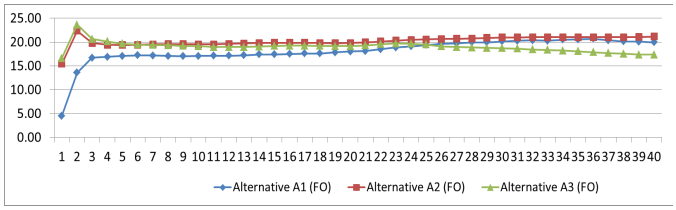

Figure 5. Fan Out of Component Candidate

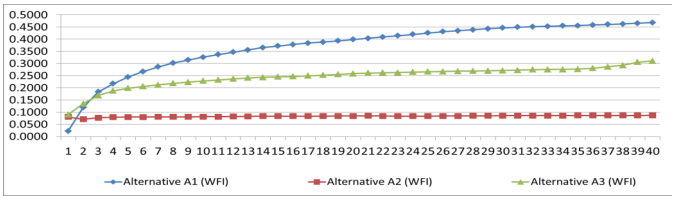

Figure 6. Fan In of Component Candidate

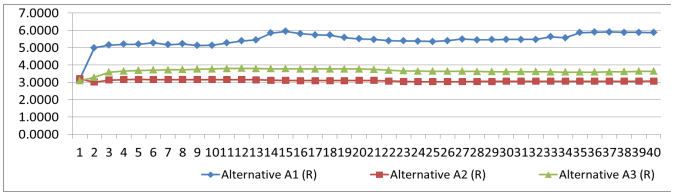

Figure 7. Reusability of Component Candidate

In Table 12 we present the percentage of cases when each class selection alternative is the optimum selection strategy, w.r.t fan out, weighted fan in and reusability. A graphical representation of the same information taking in account the size of the component retrieved by Alternative A1 is presented in Figures 8 to 10. In the line charts the x-axis represents the size of components, whereas the y-axis of the graphs represents the count.

Table 12. Optimum selection Alternative

	Fan Out	Fan In	Reusability
Alternative A1	37.8%	69.2%	64.2%
Alternative A2	13.7%	0.6%	15.3%
Alternative A3	30.3%	29.1%	13.7%
Tie	18.1%	1.1%	6.7%

The results suggest that in the majority of cases, Alternative A1 indicates the optimum set of classes that should be reused w.r.t all metrics considered.

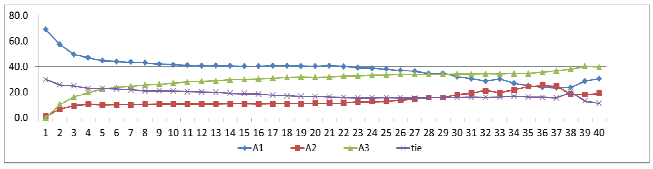

Figure 8. Optimum Selection Alternative Frequencies (Fan Out)

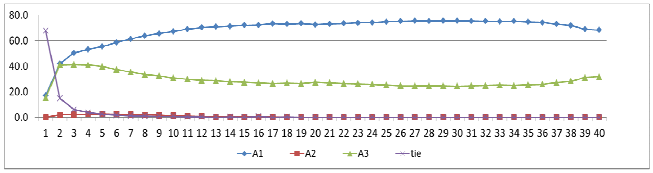

Figure 9. Optimum Selection Alternative Frequencies (Fan In)

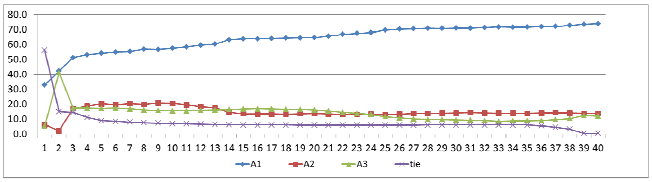

Figure 10. Optimum Selection Alternative Frequencies (Reusability)

From the above figures we can observe that there are class set size thresholds, where the proposed methodology retrieves the optimum candidate components. Concerning FO, the proposed methodology presents optimum results for components size less than twenty nine (29) classes. Additionally, concerning FI and R, the proposed methodology is the optimum solution for components size more than three (3) classes. Thus, if the size of the selected set of classes is between 3 and 29, the reuser should prefer Alternative A1for selecting the set of classes to be reused

4. SEARCH ENGINE

In order for the results of this study to be available and easily applied to practice we have registered the extracted components in an existing component web repository. The search engines of the repository enable the user to search for a specified component, and filter the result set retrieved w.r.t. component size, external dependencies, functionality and reusability. Some screenshots of the repository are presented in Figures 11 and 12.

5. ILLUSTRATIVE EXAMPLE

Let a developer of a Risk game who wants to implement the features related to the functionalities of Countries. The developer uses the search engine of Percerons and sorts the results w.r.t. size and external dependencies. The metric scores for the selected component (**Number of Classes:** 6, **External Dependencies:** 0, **Functionality:** 22, **Reusability:** 8.8283) are better than the corresponding package, i.e. net.yura.domination.engine.core (**Number of Classes:** 7, **External Dependencies:** 2, **Functionality:** 7, **Reusability:** 9.56). Suppose that one developer wants to setup a Risk game with two human players and one computer player, with their initial regional settings.

The extracted component provides functional requirements such as *country* and *continent* manipulation, i.e. defines boundary regions, region capitals, military resources, and mission control.

The component is neither involved in the game mechanics of a Risk game nor with the graphical representation of the world. Thus, the reuser can use it with his own game mechanics, such as offensive and defensive rules, and GUI. Additionally, the component updates automatically the statistics that are related to each player and the game world. The source code of a sample component execution scenario is presented in Figure 13. The class diagram of the proposed component is presented in Figure 14.

6. CONCLUSIONS

This paper introduces a methodology that can be used for extracting candidate software components from open source games. The component extraction method is based on class dependencies, in order to minimize the number of external dependencies of the candidate component. The methodology has been evaluated by comparing 577.319 candidate components with corresponding software packages. In the majority of the cases, the components extracted with the proposed methodology seem to provide minimum external dependencies, maximum functionality and maximum reusability. In order for the extracted components to be helpful for software developers, they have been recorded in

an online web repository, which makes them available, through a web search interface.

7. REFERENCES

[1] A. Ampatzoglou, I. Stamelos, "Software engineering research for computer games: A systematic review", *Information and Software Technology*, Elsevier, 52 (9), pp. 888-901 (2010)

[2] A. Ampatzoglou, A. Kritikos, G. Kakarontzas and I. Stamelos, "An Empirical Investigation on the Reusability of Design Patterns and Software Packages", *Journal of Systems and Software*, Elsevier, December 2011, volume 84, pp. 2265-2283.

[3] J. Bansiya, C. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment", *Transaction on Software Engineering*, IEEE Computer Society, 28 (1), pp. 4-17, 2002.

[4] Consumer Electronics Association, "Digital America", published electronically at http://www.ce.org.

[5] H. Cho and J.S. Yang, "Architecture patterns for mobile games product lines", *Proceedings of the 2008 International Conference on Advanced Communication Technology (ICACT'08)*, IEEE Computer Society, pp. 118-122, Phoenix Park, Korea, 17 – 20 February 2008.

[6] E. Folmer, "Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines", *10th International Symposium on Component Based Software Engineering (CBSE' 07)*, Springer-Verlag, pp. 66-73, Medford, MA, USA, 9 – 11 July 2007

[7] M. Furini, "An architecture to easily produce adventure and movie games for the mobile scenario", *Computers in Entertainment*, Association for Computing Machinery, 6(2), pp. 1-16, July 2008

[8] E. Gamma, R. Helms, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", *Addison-Wesley Professional*, Reading, MA, 1995

[9] W.P. Lee, L.J.Liu, J.A.Chiou, "A Component-Based Framework to Rapidly Prototype Online Chess Games for Home Entertainment", *Proceedings of the International Conference on Systems, Man and Cybermetrics (SMC'06)*, IEEE Computer Society, pp. 4011 – 4016, Taipei, Taiwan, 8-11 October 2006

[10] R. C. Martin, "Agile Software Development: Principles,Patterns and Practices", *Prentice Hall*, Upper Saddle River, NJ, 2003

[11] M. McShaffry, "Game Coding Complete", *Paraglyph Press*, Arizona, USA, 2003

[12] E.B. Passos, J. Weslley, E. Walter G. Clua, A. Montenegro and L. Murta, "Smart composition of game objects using dependency injection", *Computers in Entertainment*, Association for Computing Machinery, 7(4), October 2009

[13] C. Szyperski, "Component Software: Beyond Object-Oriented Programming", *Addison-Wesley International*, Massachusetts, USA, 1997.

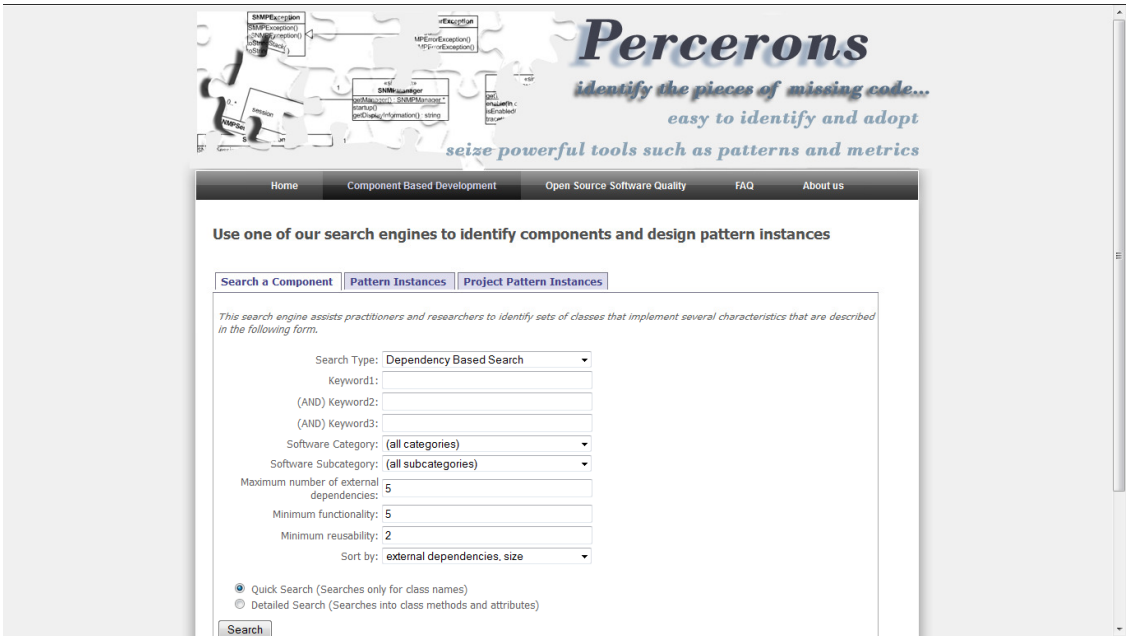

Figure 11. Web Repository Search Screen

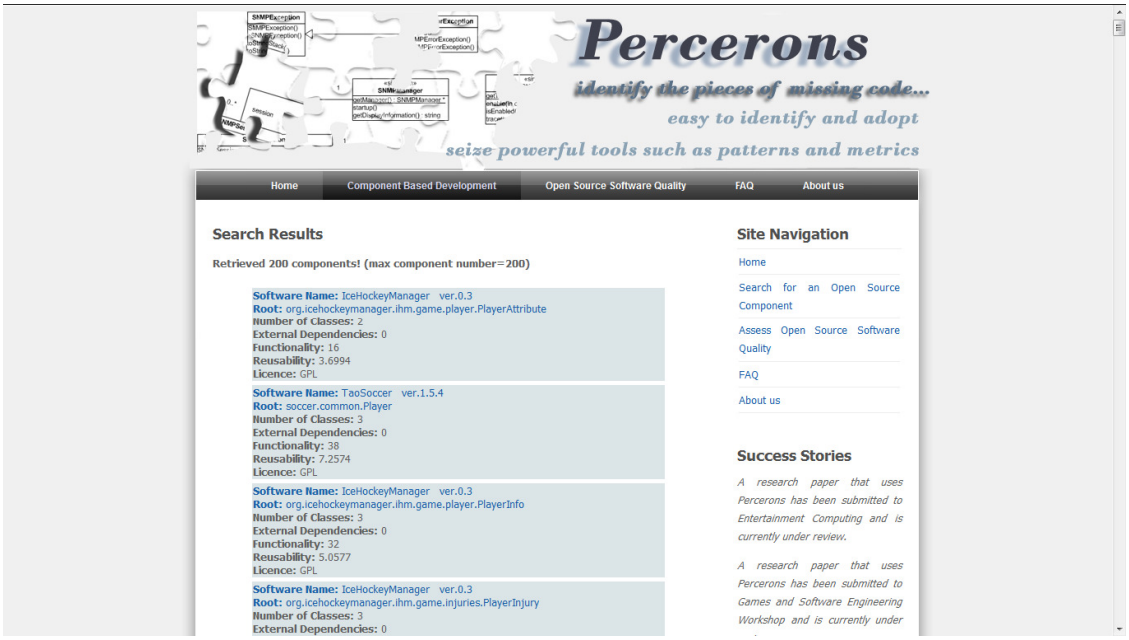

Figure 12. Web Repository Search Screen

```
public class RiskExecutionScenario {

    public static void main(String[] args) {
        Continent europe = new Continent("1", "Europe", 10, 1);
        Country france = new Country(1, "1", "France", europe, 0, 0);
        Country spain = new Country(2, "2", "Spain", europe, 15, 15);
        Continent asia = new Continent("2", "Asia", 15, 2);
        Country india = new Country(3, "3", "India", asia, 50, 50);
        Country china = new Country(4, "4", "China", asia, 40, 100);
        Continent africa = new Continent("2", "Africa", 100, 0);
        Country egypt = new Country(5, "5", "Egypt", africa, 100, 10);
        Country algeria = new Country(6, "6", "Algeria", africa, 100, 50);

        Card card1 = new Card("Infantry", india);
        Card card2 = new Card("Cannon", france);
        Card card3 = new Card("Cavalry", spain);
        Card card4 = new Card("Infantry", china);
        Card card5 = new Card("Infantry", egypt);
        Card card6 = new Card("Cavalry", algeria);

        Player player = new Player(1, "angor", 1, "playerladdress");
        player.addArmies(5);
        player.setCapital(france);
        Mission mission = new Mission(player, 6, 1, europe, asia, africa, "Conquer the world!");

        Player player2 = new Player(1, "apostolos", 2, "player2address");
        player2.addArmies(5);
        player2.setCapital(china);
        Mission mission2 = new Mission(player2, 6, 1, europe, asia, africa, "Conquer the world!");

        Player player3 = new Player(0, "bot", 3, "player3address");
        player3.addArmies(3);
        player3.setCapital(algeria);
        Mission mission3 = new Mission(player3, 6, 2, europe, asia, africa, "Conquer the world!");
        }
}
```

Figure 13. Country Component Execution Scenario

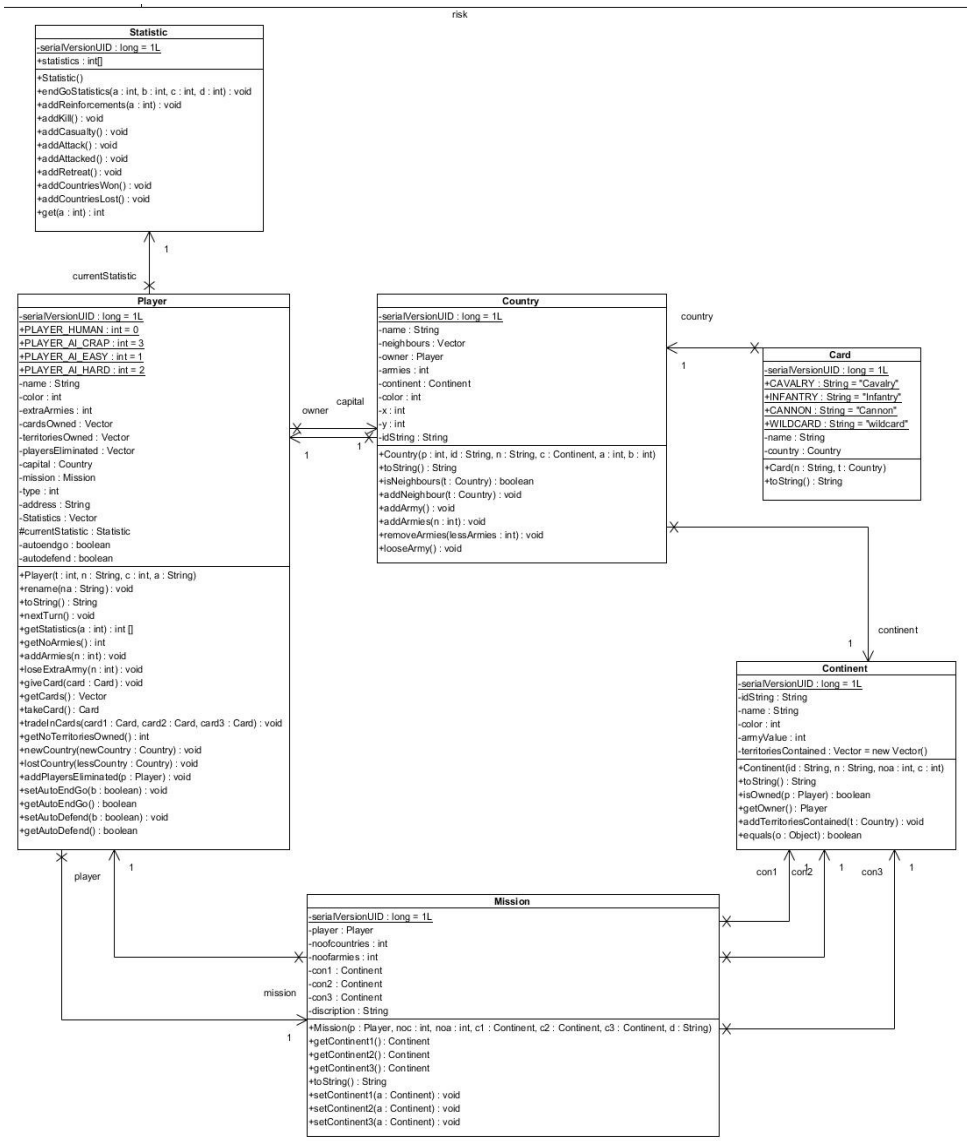

Figure 14. Continent Component Class Diagram