Software Product Line Extraction from Variability-Rich Systems: The Robocode Case Study

Jabier Martinez, Xhevahire Tërnava, Tewfik Ziadi

Software Product Line Extraction from Variability-Rich Systems: The Robocode Case Study

Jabier Martinez*
Tecnalia
Derio, Spain
jabier.martinez@tecnalia.com

Xhevahire Tërnava
Sorbonne University UPMC
Paris, France
xhevahire.ternava@lip6.fr

Tewfik Ziadi
Sorbonne University UPMC
Paris, France
tewfik.ziadi@lip6.fr

ABSTRACT

The engineering of a Software Product Line (SPL), either by creating it from scratch or through the re-engineering of existing variants, it uses to be a project that spans several years with a high investment. It is often hard to analyse and quantify this investment, especially in the context of extractive SPL adoption when the related software variants are independently created by different developers following different system architectures and implementation conventions. This paper reports an experience on the creation of an SPL by re-engineering system variants implemented around an educational game called Robocode. The objective of this game is to program a bot (a battle tank) that battles against the bots of other developers. The world-wide Robocode community creates and maintains a large base of knowledge and implementations that are mainly organized in terms of features, although not presented as an SPL. Therefore, a group of master students analysed this variability-rich domain and extracted a Robocode SPL. We present the results of such extraction augmented with an analysis and a quantification regarding the spent time and effort. We believe that the results and the a-posteriori analysis can provide insights on global challenges on SPL adoption. We also provide all the elements to SPL educators to reproduce the teaching activity, and we make available this SPL to be used for any research purpose.

CCS CONCEPTS

• **Software and its engineering → Software reverse engineering**; **Software product lines**; • **Social and professional topics → Software engineering education**;

KEYWORDS

Software Product Lines, Reverse-engineering, Extractive Software Product Line Adoption, Education, Robocode

1 INTRODUCTION

Software Product Line (SPL) engineering enables a systematic reuse within a family of systems. Its general framework is defined by the dual phases of domain and application engineering [2]. During domain engineering, the scope of the SPL is defined, and *commonalities* and *variabilities* among software products are explicitly specified. The prime entities in this specification use to be based on the concept of *feature*, which is defined as a prominent or distinctive characteristic, quality or user-visible aspect of a software system or systems [9]. Domain engineering also aims at implementing *reusable assets* and shaping them according to the identified features. Domain engineering can be achieved using an extractive SPL adoption approach [11] where existing legacy artefact variants are analysed and their variability models with reusable assets are extracted. The existence of legacy artefact variants is a usual case in SPL adoption (50% of the cases according to an industrial survey [5]), and re-engineering them into an SPL is still a challenging process requiring a high investment which is often difficult to analyse and quantify. In addition, the problem is exacerbated when the related software variants are not created through the clone-and-own approach but independently created by different developers following different system architectures and implementation conventions.

In this work, we apply the extractive SPL adoption approach to a variability-rich family of systems, the Robocode programming game[1] and, more specifically, the source code-based bots implemented by its community. Robocode has a large body of documentation and available bot implementations, which make it an appealing domain for our study. Moreover, it is not organized as an SPL, as different developers create independently their own bot variants for their different needs. The objectives of our work are threefold: (1) to report an experience on the extraction of an SPL which can help to bring light to SPL extraction issues in a comprehensible scenario, (2) to analyse and quantify the investment required to extract the SPL from the existing artefact variants, (3) to contribute pedagogical material on SPL adoption which is a real industrial need [15] not sufficiently covered by current pedagogical material [1]. A group of students had the task to analyse the Robocode domain, formalize the bots commonalities and variabilities using feature modelling [9] in the FeatureIDE tool [21], and to extract feature-based reusable assets using a compositional approach to implement variability based on the FeatureHouse composer [3].

We present in this paper the results of the extracted Robocode SPL in terms of features and their associated reusable assets. We discuss the time and effort spent during the SPL extraction based on the collected answers from a survey that we conducted among the participants. Principally, we present the distribution of time in the different activities and discuss some reflections on the gained knowledge. In addition, we make publicly available the extracted SPL and the teaching material, and we present how the extracted SPL can be used beyond education. The findings are not only interesting for educators but also for a wider audience interested in understanding SPL extraction issues from a practical perspective. These issues appear during the end-to-end extraction of our "academic-scale" family of systems. Moreover, the provided material together with our publicly available extracted Robocode SPL can be used to reproduce the case study and compare with our results.

*The work of Jabier Martinez was mainly done during his stay at Sorbonne University UPMC as post-doctoral researcher.

[1] Robocode: http://robocode.sourceforge.net

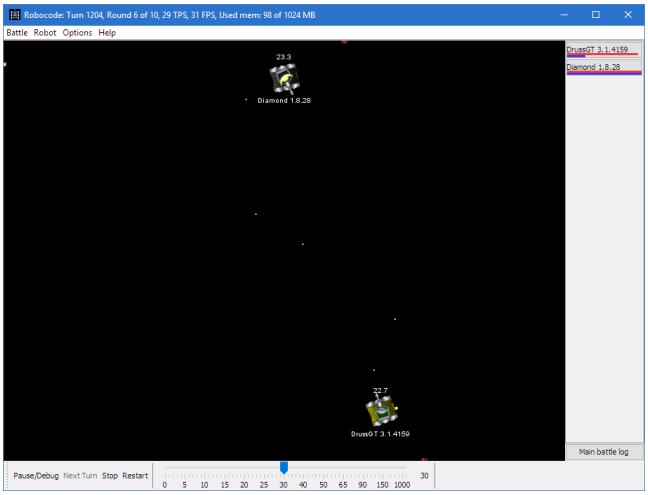

Figure 1: Battle of two bots. Screenshot from the RoboWiki

The Robocode SPL including technical information for how to use it, and extra material is available [2].

In the following, we present background information about Robocode and its existing assets (Section 2). We also present the case study settings including the participants' description and the context and duration of the project (Section 3). We report the results of the domain analysis phase (Section 4) and the results of the domain implementation phase (Section 5). Further, an analysis and discussion of the results are also given (Section 6). We then outline some possible uses of our extracted Robocode SPL (Section 7). Finally, we present related works (Section 8) and we conclude the paper while outlining future work (Section 9).

2 BACKGROUND ON ROBOCODE

An API of the Robocode game, in Java or .NET, is provided to any participant to develop a virtual battle tank (bot from now onwards) that battles against other bots from other developers in a closed-arena. A battle simulation environment is provided to observe the programmed behavior and debug the bots as shown in the screenshot of Figure 1. Under the motto "Build the best, destroy the rest", the Robocode community is active with a long history which started in 2001. Besides its pedagogical values for teaching programming [18] or even artificial intelligence approaches [8], it has been also used as case study for advanced computational intelligence research (*e.g.*, genetic programming [20]).

Game rules. A Java API [3] is provided to implement a virtual bot with body, gun, and radar (as shown in Figure 2). We explain some of the most relevant API methods. The body will move with the `setAhead` method that takes a distance as parameter. Also the desired velocity can be defined. The body can be turned with `setTurnRight` and a desired angle. In a similar way, the gun, which is on top of the body, can be turned with `setTurnGunRight`. One can also decide when to fire with `setFireBullet` and defining as parameter which power they want to put in the bullet (bullets with lower power travel faster, and our bot will lose less energy for firing the bullet). The radar, on top of the gun, can be turned with `setTurnRadarRight`. There are several events that can be listened by the bot, the most important one is the event when the radar detects an enemy bot because its position can be stored (bullets are not detected by the radar). Other examples of events are related to being hit by a bullet, by a wall, or by a bot.

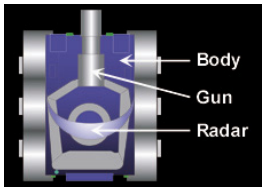

Figure 2: Anatomy of a bot. Illustration from the RoboWiki

These API primitives for movement, targeting, and radar enable to program very diverse and advanced techniques to face the battle. These techniques are behavioral features which are the more relevant source of diversity in the Robocode family of bots. There are other sources of variability such as the colors of the body, gun, radar and bullets which are not of our interest in this work as they do not have any impact on the behavior. There is also variability in the type of battle such as *1 vs 1* (see Figure 1), *melee* with several bots fighting among them, or *team* where several bots communicate among teammates and fight against another team. This variability is not taken into account as we focused on behavioral features of the bots.

Knowledge and assets base. The Robocode community has created a large knowledge base which is centralized in their official wiki called RoboWiki [4]. Beginners and advanced developers can find, share and discuss any topic with a special focus on bots' features. A plethora of bots is publicly available including its source code in common or personal repositories. For example, more than 2900 bots (including variants and versions) are available in a public repository [5]. We automatically checked that, in this repository, more than 1500 made their source code publicly available with an average of 24.38 Java files each. Around one-third of these 1500 are bots with only one or very few number of Java files. The reason is that there are Robocode competitions where the code size is restricted giving rise to the *haiku* [20], *nano*, *mini* or *micro* categories. For those that do not have code size restrictions, we can find bots with hundreds of Java files.

The Robocode community, as result of their collaborative work, maintains the documentation of known features. This includes a description and, in some cases, some hints or snippets of how to implement them. Apart from the RoboWiki pages dedicated to the features, 300 bots are documented in the RoboWiki by the authors themselves[6], including information about the features used for movement and targeting. Despite all this Robocode knowledge and assets base, there is an absence of an explicit specification of all the existing variability among these thousands of bots. The Robocode SPL extra material that we make public includes the data we gathered about the bots repository and the RoboWiki bots.

[2] Robocode SPL: https://github.com/but4reuse/RobocodeSPL-Matching
[3] Robocode Java API: http://robocode.sourceforge.net/docs/robocode/
[4] RoboWiki: http://www.robowiki.net
[5] Robocode archive: http://robocode-archive.strangeautomata.com/robots/
[6] http://robowiki.net/wiki/Category:Bots

3 THE ROBOCODE CASE STUDY

This Section explains the Module framework which was used as the reference architecture for the SPL, and the case study settings regarding the task and the involved participants.

3.1 An Imposed Reference Architecture: The Module Framework

Robocode bots are development projects requiring a certain design and structure to ease their maintenance and iterative enhancements. As mentioned in Section 2, several bots are implemented in just one Java class, especially those competing in categories with code size limitations. However, bots without these limitations use to be more carefully structured. To facilitate the reuse of common functionalities and to facilitate the integration of behavioral components, certain members of the community have developed frameworks aiming to ease the development of bots. One example is the Xander framework [7] or the Module framework [8]. In this work, we used the Module framework (version 1.0.0) which was created and documented by the first author. The UML Class diagram of Figure 3 presents an excerpt of the framework. The abstract class `Module` extends `TeamRobot` from the official Robocode API, so a Java class implementing `Module` can be used in Robocode as a new bot. The `Module` class takes care of managing all the information about the battle (*e.g.*, listening and processing events, updating enemy positions, keeping the history of fired bullets, sharing information if there are teammates, and executing the behavior by extending the `run` method from the `Robot` class). The abstract methods `initialize` and `selectBehavior` will be specialized when a bot instance wants to be created. The `Module` class contains five `Parts` and it will notify the parts for each event that happens during the battle. Each part has its own responsibility:

- `Radar`: to scan the enemies by moving the radar.
- `Movement`: to move the bot by rotating and setting the speed.
- `Targeting`: to target by turning the gun. This will usually require to use the current enemy position but not necessarily. For example, the `AreaTargeting` strategy will turn the gun to the area of the battlefield with the greatest number of enemies.
- `Gun`: to set the fire power.
- `SelectEnemy`: to select the current main enemy among all the enemies.

[7] http://robowiki.net/wiki/XanderFramework
[8] https://github.com/jabiercoding/ModuleRobocodeFramework

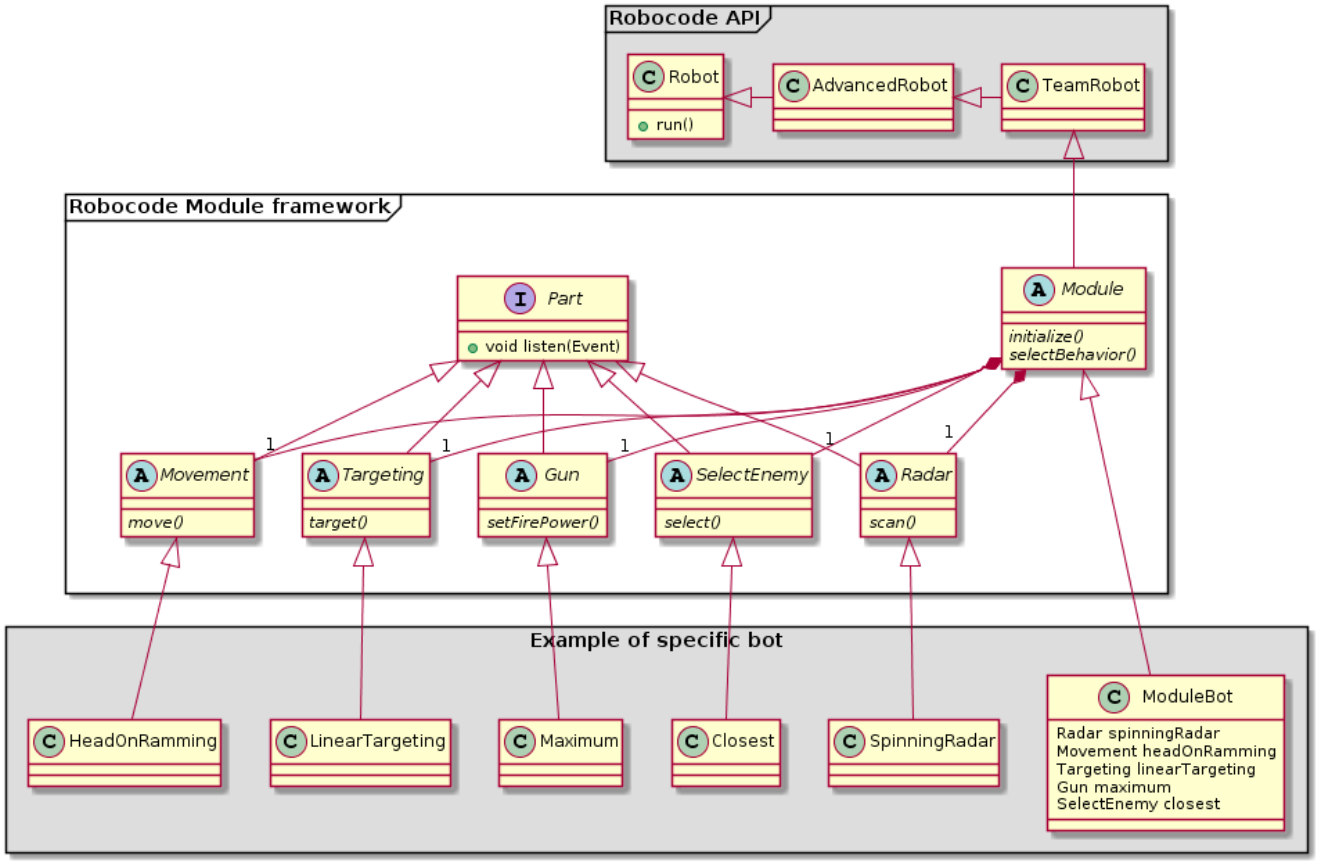

Figure 3: Excerpt of a UML Class diagram illustrating the Module framework, its relation with the Robocode API, and an example of a bot implementing a Module bot

At the bottom of Figure 3, we have an example of a bot named `ModuleBot`. This bot extends `Module` and the behavior is defined with the implementation of the different `Parts`. `SpinningRadar` is a radar behavior that turns the radar indefinitely without focusing on any specific enemy. `HeadOnRamming` is a movement that tries to crash the enemy by continuously moving towards the enemy position at the maximum speed. `LinearTargeting` is a targeting behavior that assumes that the enemy will continue in the same direction and at the same speed to estimate its position and target the gun accordingly. `Maximum` is a gun behavior to always fire with the maximum of bullet power. `Closest` is a `SelectEnemy` behavior that continuously checks which is the nearest enemy to select it as the current enemy.

The Module framework follows the strategy pattern to include the bot behavior so we considered that it was a good candidate to be used as the base for the implementation of an SPL. This imposed framework for the SPL required that features extracted from the RoboWiki or from existing bots will need to conform to the framework. In general, the integration in the framework was not difficult (as stated by the SPL developers), but they were aware of this activity during the re-engineering of the existing code.

3.2 Case Study Settings

Participants and duration. The case study included 6 Master students at Sorbonne Université (last year before starting a Ph.D. or starting an industrial professional career) over 3 months to extract the Robocode SPL. The six participants (three men and three women, uniform in age and in their university background) formed a group in the context of a course related to project management. The technical tasks for software development within the course must not be trivial but challenging for the three months of its duration. The Robocode SPL was realized in this time frame, working as a self-organized group, and in parallel with other courses and projects in their master program.

Training activities. The participants had no initial knowledge of SPL theory nor on any tool for SPL implementation. In view of this project, a tutorial was conducted during 4 hours by the last author of this paper, as a senior expert on SPLE. This tutorial included an introduction to SPLs and a hands-on practice on FeatureIDE and FeatureHouse. In a similar way, they had not any knowledge on Robocode. Another 4 hour tutorial was performed to teach Robocode and the Module framework by the first author of this paper. A small example of FeatureIDE usage to derive variants of bots was implemented. This SPL was then used as a base to incrementally develop the final Robocode SPL. Despite that their work was autonomous, we gave support for specific questions about the SPL implementation tooling and Robocode.

4 DOMAIN ANALYSIS

Formalization of domain variability. Figure 4 shows the resulting feature model [9]. Feature names are not readable but it serves to get an impression of its size and topology. The feature model includes a total of 115 features from which 7 are part of the mandatory core. 22 are abstract features intended to group alternative or optional features, and the remaining 93 are behavioral features intended to have an implementation counterpart. Features starting from level two of the feature model tree are behavior techniques which are grouped as alternative features (*i.e.*, only one can be selected). For example, for the `Radar`, there are 9 feature alternatives, or for the `Movement`, there are 27 alternatives grouped in different movement categories. On the zoomed part of Figure 4, we can see 5 out of these 9 features related to radar, and we can also see a targeting category called `BasicTargeting` with 7 alternative features. Given the combinatorial explosion of possible configurations, the number of possible variants exceeds several hundred thousands. The FeatureIDE tool was not able to compute an exact number after several hours of computation.

There are only two exceptions where the features are not grouped as alternatives but as a set of optional features (*i.e.*, you can select none of them or any of them). One of them are features related to `EnergyManagement` techniques. When a bot fires a bullet, the bot loses a certain energy amount depending on the power that was established for the bullet. Energy management is a set of optional

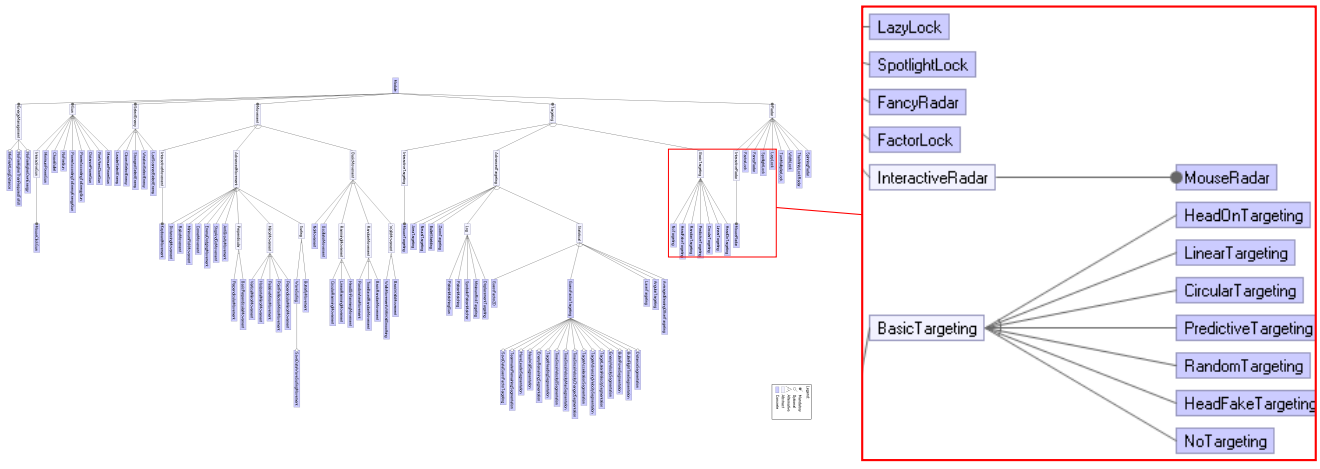

Figure 4: An excerpt of the extracted feature model for Robocode SPL

features to preserve the energy of the bot under certain circumstances. For example, if our bot energy is low, some bots establish the condition that we will never fire with a power value equal or higher than the value that will disable our bot. In those cases where the enemy energy is low, some bots reduce the bullet power to the strictly necessary power to eventually destroy the target. The other set of optional features are related to the `GuessFactorTargeting` feature that targets based on the histograms of the angles of previously fired bullets. These histograms can be segmented based on some battle information. For example, different histograms based on the enemy distance (*e.g.*, when it is close, when it is far) or based on the enemy velocity or acceleration (*e.g.*, when it is stopped, when it is moving). More than 10 segmentation options are added as optional features. The implementation of concrete examples will be shown in Section 5. Apart from these alternative constraints and groups of optional features, there are no cross-tree constraints.

Representativeness. It is difficult to measure how representative is this feature model to the existing variability in the Robocode domain which is quite vast. Based on our experience, we consider that it is representative of the main topics of the domain but we are aware that several concepts are missing. Examples of non-covered topics are: Droid bots are special bots with no radar, features to perform file-based serialization of data as bots can optionally store data of previous matches (*e.g.*, `GuessFactorTargeting` histograms), or features to include or exclude debugging graphics of the features.

5 DOMAIN IMPLEMENTATION

Technical solution. Features in a variability model need its implementation counterpart to derive actual product variants. As variability implementation mechanism, we decided to impose the FeatureHouse [3] composer available in FeatureIDE [21] which enables the composition of Java source code. Annotative approaches such as the preprocessor-based Antenna or Munge (Java-based alternatives to the widespread usage of *#ifdef* pre-processor directives in C source code) were also available in FeatureIDE but we opted for FeatureHouse to force a more clear separation of features.

To present a feature implementation and its composition we used a basic targeting feature called `HeadOnTargeting`. This technique consists in aiming the gun directly to the enemy position (or at least where we last saw our enemy). We use the notation used by Apel et al. [3] where the symbol '•' is used to denote the composition operator of software artefacts. Figure 5 illustrates the composition of the feature structure trees (FSTs) [3] of the `HeadOnTargeting` and `Module` features. The composition operation results in an FST where

```
1  package jab.targeting;
2
3  public class HeadOnTargeting extends Targeting {
4
5    public HeadOnTargeting(Module bot) {
6      super(bot);
7    }
8
9    public void target() {
10     if (bot.enemy != null) {
11       double absoluteBearing = bot.getHeadingRadians() +
       bot.enemy.bearingRadians;      bot.
       setTurnGunRightRadians(robocode.util.Utils.
       normalRelativeAngle(absoluteBearing - bot.
       getGunHeadingRadians()));
12     }
13   }
14 }
```

```
1  package jab;
2
3  public class ModuleBot extends Module {
4    Targeting selectedTargeting = new HeadOnTargeting(this)
       ;
5  }
```

Listing 1: The implementation of feature `HeadOnTargeting`

the `HeadOnTargeting` feature is included in a bot variant. Listing 1 shows the complete source code of the `HeadOnTargeting` feature. The first Java class `HeadOnTargeting` extends the `Targeting` class of the Module framework and the `target` method is implemented to turn the gun. The other Java file below this one is a fragment of source code that just declares the variable `selectedTargeting` and assigns it an instance of the `HeadOnTargeting` class. In Listing 2 we show how the `HeadOnTargeting` is composed using FeatureHouse inside the `Module` mandatory feature. In line 5 of Listing 2, we can observe how the variable coming from line 4 in the second part of Listing 1 was added. Then, if the `HeadOnTargeting` feature is selected, the instance of `HeadOnTargeting` will be the targeting behavior of the derived bot variant.

Size metrics. The extracted Robocode SPL had 80 features with an actual implementation counterpart. Not all the 115 features in the feature model had an implementation, as mentioned in Section 4, 22 of them are abstract features. Also, the students did not have time to implement 13 features identified in the domain analysis given the constrained duration of the project. Figure 6 shows, in the first box-plot on the left, the distribution of the

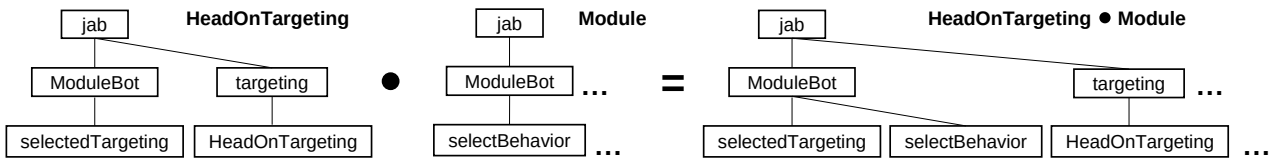

Figure 5: `HeadOnTargeting` • `Module`. The composition of the feature `HeadOnTargeting` in the `Module` mandatory feature. (The notation is adapted from Apel et al. [3])

```
package jab;

public class ModuleBot extends Module {
  ...
  Targeting selectedTargeting = new HeadOnTargeting(this)
     ;
  ...
  protected void selectBehavior() {
    radar = selectedRadar;
    movement = selectedMovement;
    targeting = selectedTargeting;
    selectEnemy = selectedSelectEnemy;
    gun = selectedGun;
  }
    ...
}
```

Listing 2: `HeadOnTargeting` • `Module`. The composition of the feature `HeadOnTargeting` in the `Module` mandatory feature

number of lines of code (LoC)[9] for the 80 features. The median is 30 LoC but we can observe several features with higher numbers. The outlier counting 633 LoC, in the upper left of the figure, corresponds to the mandatory `Module` feature (*i.e.*, the root of the feature model) that, as described in Section 3.1, is responsible to manage all the data of the battle and handle the events before dispatching them to the bot parts. The rest of the box-plots show the distribution of subsets of features corresponding to the six main relevant feature categories: `Targeting`, `Movement`, `Gun`, `SelectEnemy`, `Radar`, and `EnergyManagement`. We can observe that, in general, movement features are the category with a higher number of LoC, followed by targeting features. On the contrary, the features related to energy management, radar, enemy selection or gun are rather small. The other outlier of the "All features" box-plot with more than 400 LoC corresponded to a movement feature. This feature is called `WaveSurfing` and it is an advanced technique used by many high-competitive bots [10]. There are also tiny feature implementations. Two of the smallest features with an implementation are related to `EnergyManagement` features such as `NoFireHigherThanRequiredToKill` and `NoFireHigherOwnEnergy`, which implementation will be explained and shown in Listing 3.

The Robocode SPL implementation size, measured as the sum of the feature implementations, is 4,403 LoC. The mandatory `Module` feature represents already the 14.38%. Then, the features under `Targeting` and `Movement` constitute the 71.43%, and 14.19% for the rest of the features all together. The size of the SPL is not realistic in industrial SPLs. However, in the Robocode domain, these feature sizes are realistic. Also, we consider that the relatively small size of the features enables to implement a large diversity of features within the time constraints of the project. This characteristic makes Robocode a good candidate for educational purposes.

Feature interactions. A feature interaction is some way in which a feature or features modify or influence another feature in describing or generating the system's overall behavior [22]. In the Robocode SPL, there are two cases of feature interactions, (1) the `EnergyManagement` features modifying the result of the `Gun` features in the assignation of the bullet power,

[9]LoC calculated with Google CodePro Analytics
[10]http://robowiki.net/wiki/Wave_Surfing

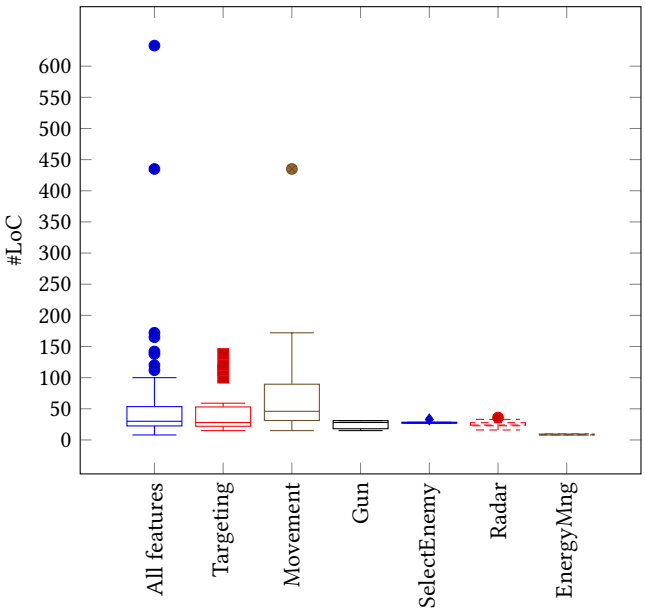

Figure 6: #LoC for all features, and according to the six main compound features

```
package jab.module;

public class Module {
  protected void energyManagement() {
    original();

    if (enemy!=null){
      bulletPower = Math.min(bulletPower, getEnergy() -
     0.01);
    }
  }
}
```

Listing 3: The implementation of feature `NoFireHigherOwnEnergy`

and (2) the segmentation features influencing the behavior of the default `GuessFactorTargeting` feature. To implement the `EnergyManagement` features, the participants evolved the Module framework to support this new concept which is complementary to the `Gun` strategies. An `energyManagement` method was added to the abstract `Module` class where the default behavior is not to perform any kind of energy management and takes directly the power defined by the `Gun` feature. Then, the optional energy management features, if selected, will chain conditions in this method to preserve energy. Listing 3 shows the implementation of the feature `NoFireHigherOwnEnergy`. In Line 5 we can see a call to `original()`, which is a FeatureHouse reserved method to include the original code of the feature where this feature will be composed.

The `GuessFactorTargeting` feature was designed in a way that optional features can define the segment that applies at each moment in the battle. Listing 4 shows the implementation of `DistanceSegmentation` that interacts with `GuessFactorTargeting` by defining the segment that applies according to the distance to the current enemy. In Line 17 we can

```
package jab.targeting;

public class GuessFactorTargeting extends Targeting {

  public void target() {
    if(bot.enemy!= null) {
      if (bot.enemy.distance <= 100) {
        segmentsValues.add("close");
      } else if (bot.enemy.distance > 100 && bot.enemy.
     distance <= 300) {
        segmentsValues.add("less close");
      } else if (bot.enemy.distance > 300 && bot.enemy.
     distance <= 500) {
        segmentsValues.add("far");
      } else {
        segmentsValues.add("very far");
      }
    }
    original();
  }
}
```

Listing 4: The implementation of feature `DistanceSegmentation`

see the call to the `original()` method that will include the original code of `GuessFactorTargeting`. This advanced technique in the use of FeatureHouse was the solution they found to include segments. In general, different implementation solutions can be found to solve diverse technical problems in SPL implementation. We did not compare their solution with other possibilities but they proposed working solutions which we consider valid.

Duplicated code in feature implementations. The a-posteriori analysis of the implementation revealed several cases of duplicated code among some features. For example, a method `goTo(x,y)` for moving the bot to a given battlefield position is present in more than ten features. The SPL features were not refactored to include these clones in separated features (that will be included if any of the features were selected). Moreover, this will avoid having source code clones among features but there is the trade-off that features will not be self-contained. There are still discussions about when cloning is considered harmful [10], but in the Robocode SPL, the identified clones can represent issues in maintenance. For example, at least three different ways to implement the same `goTo` method can be found in the resulting SPL.

6 RESULTS, DISCUSSIONS AND LIMITATIONS

This section discusses the results from different perspectives, (1) its educational value, (2) the extractive process, and (3) the analysis of the time and effort. The results are based on individual questionnaires (available in the Robocode SPL extra material) and interviews.

6.1 Educational Value

The results of the knowledge gained during the project are shown in Figure 7. Concretely, it shows the average of their self-assessment before and after the project on a scale from one to ten. We can observe how their knowledge in the Java programming language remains almost constant while more advanced software engineering concepts such as feature modeling, implementing feature-based systems and SPLs are highly increased. With almost the same increase, we find the understanding of the Robocode domain and the technical skills required to use the FeatureIDE tool. One conclusion that can be obtained from this graph is that this experience has no knowledge pre-requirements for the participants apart from knowing Java (a minimum requirement accessible to anyone with basic knowledge of programming). The result is a dramatic gain of knowledge to design and implement an SPL. Compared to the traditional single-system development, the participants gained hands-on practice on implementing systematic reuse so they can respond to more advanced software engineering challenges in implementing families of systems.

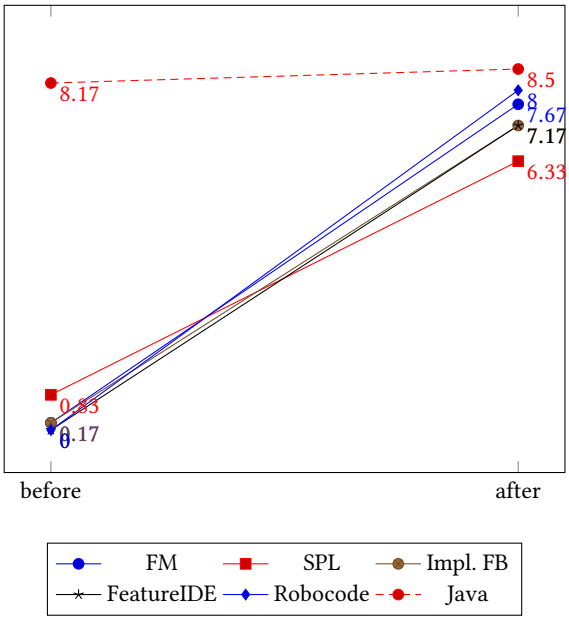

Figure 7: Reflection on the gained knowledge: Feature Modeling (FM), SPL, Implementing Feature-Based systems (Impl. FB), FeatureIDE, Robocode domain, and Java

Their high self-assessment after the project needs to be put in perspective with the scope of the project, in the sense that if they will be confronted to an industrial SPL in the future, or they will research on these topics, they will realize how broad SPL engineering is (*e.g.*, economic models, SPL scope, different ways to implement variability, different binding times etc.) and they will probably have to re-evaluate their self-assessments. However, they gained the minimum knowledge required to start using state of the art methods and tools, instead of ad hoc solutions.

6.2 Emerging SPL Extraction Process

Given the Robocode SPL extraction task, we wanted to understand how they proceeded to implement features to investigate if their process could be generalized. Figure 8 shows the process followed by the participants to implement a feature. The flow-chart is the result of individual interviews where, quite unanimously, they agree in the presented steps. Therefore, this is the process that emerged for the majority after interviewing them once the project had been completed. Each of them begins by selecting a feature that is not already implemented or assigned to others. Then, they analyse the feature information from the RoboWiki. Whenever there is a

corresponding implementation snippet for that feature, they take and adapt it to their extracted core-code assets in the Robocode SPL, while the adaptation includes the integration in the Module framework. When there is no implementation in the RoboWiki, then they have to identify which bots have this feature, so they can take the implementation from one of those bots and adapt it. Otherwise, they implemented that feature from scratch by using only its documentation. Finally, they test and debug the feature.

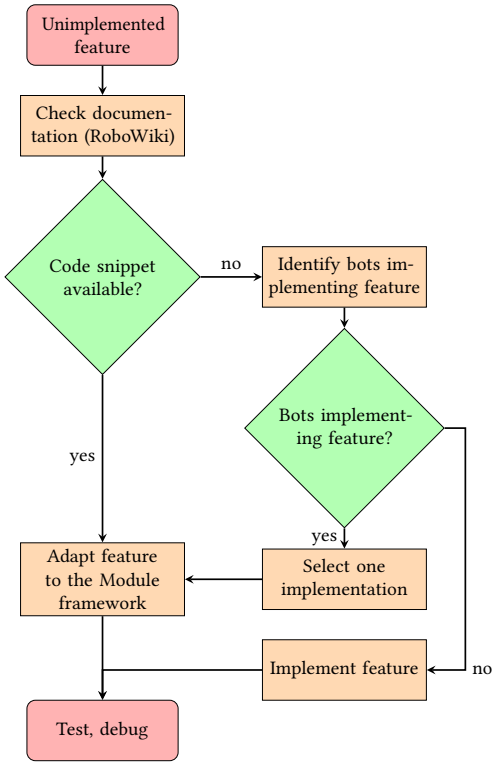

Figure 8: Feature implementation process

Following this process, the source of the implementation can be either from a source code snippet available in the RoboWiki, from an existing bot, or from the documentation in the RoboWiki. Figure 9 shows the percentage of the source of the implementation. We can observe that more than a half were taken from source code snippets available in the RoboWiki and then adapted to the Robocode SPL. Taking and adapting features from a bot was also the case for around a quarter of the features. As a last option, implementing from scratch based on the documentation also happened for 22.60% of the features.

The high percentage of the use of snippets from the RoboWiki shows that the Robocode community not only explains the principles of a feature, but there is also a will to facilitate its reuse in practice. These snippets, at a smaller scale, can be analogous to the existence of reusable components in industrial settings that can be taken to implement a feature offered by this component (*e.g.*, in-house reusable components or COTS). Taking the source code of a bot is similar to trying to locate and extract a specific feature from a complete system or from a system variant that we

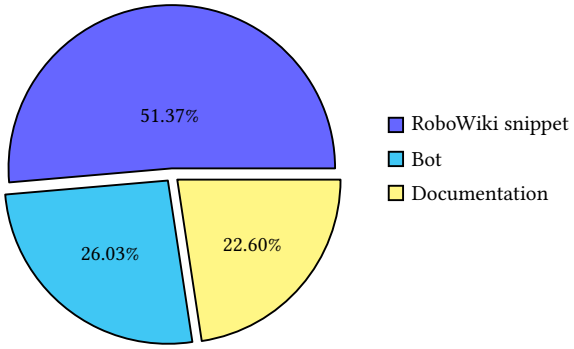

Figure 9: Sources to implement features: Robowiki code reused and adapted for the Robocode SPL, existing bot code reused and adapted for the Robocode SPL, and features implemented from scratch

know that contains this feature. Finally, implementing a feature from scratch is the case where there are no available assets to mine the feature and it has to be implemented as per specification. Given that features are documented, this case study lacks a scenario (usual in industrial settings [15]) consisting in the need to perform interviews and workshops with different technical and non-technical stakeholders to identify and understand features.

The quality of the extracted SPL artifacts was briefly discussed during the paper. The feature model is representative according to our expert judgment but it is not complete (Section 4) and duplicated code was found in feature implementations (end of Section 5). Further work will be needed to investigate which metrics can be used to evaluate the quality of the extracted SPL and which tools can be used to obtain them. More in-depth analysis of the SPL quality was out of the scope of this work.

6.3 Time and Effort During SPL Extraction

Each student was asked regarding the time spent in five main activities: (1) understanding feature modeling (FM), (2) understanding how to use FeatureIDE, (3) understanding the Robocode domain, (4) modeling the Robocode domain variability, and (5) implementing the features of the Robocode SPL. Figure 10 shows the total time of the project and the percentage dedicated to the different activities. Around half of their time was spent in the feature implementation part, which includes the time to understand certain mathematical concepts and algorithms behind the implementation of the bot behaviors. With 17.83% we have the analysis of the Robocode domain. They made a remark about several cases where features with different names were similar and they spent time thinking if they were actually different or the same. The same time was spent to understand FMs and FeatureIDE (8.67%), and a little bit less (5.67%) to actually use FeatureIDE to model the Robocode commonality and variability. Finally, an average of 11.67% of their time was spent in other activities, such as project management and documentation. In the Robocode SPL extra material we add the documentation of each feature.

Features which were well-documented in the RoboWiki and that included a snippet were much faster to extract than using

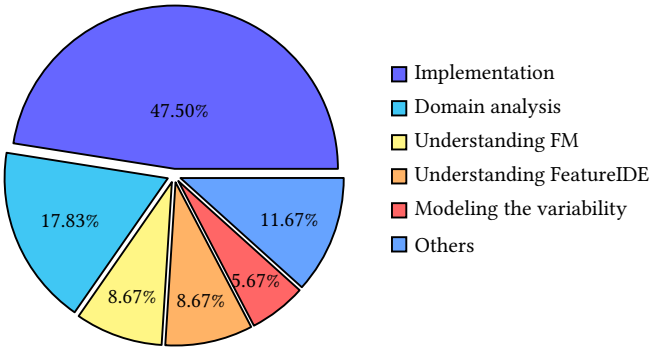

Figure 10: The average distribution of time in different activities by each participant, in percentage (%)

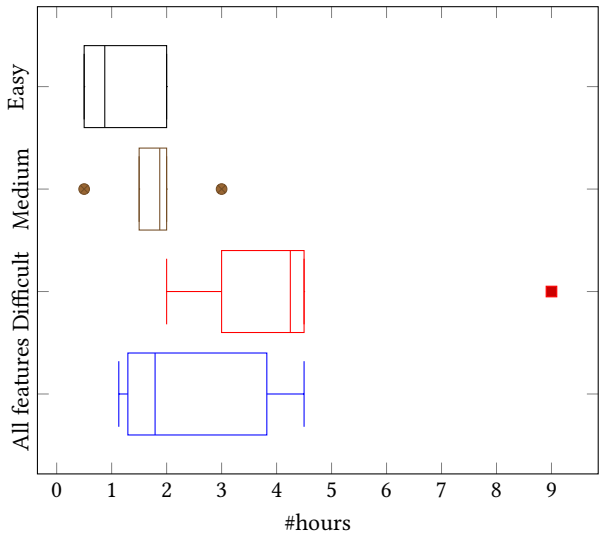

Figure 11: The time to implement a feature (easy, medium, difficult), and in average for all of them

other means, such as taking from an existing bot or implementing from scratch. This also explains why they always began the implementation process of a feature by searching it in the RoboWiki (*cf.* Figure 8), and most of the features are extracted from there (*cf.* Figure 9). The case of taking the implementation from existing bots was more complicated. Specially because several bots implement the same feature in different ways and by different developers. Thus, it was not always easy to define a criteria to know which one should be used as base for the extraction.

As the implementation or extraction of features took most of the time, they were asked for the time spent to implement a feature. First, they grouped features into three main categories (1) easy, (2) medium, and (3) difficult to implement. And, depending on these categories, they provide different implementation time for such features. Figure 11 shows in average their evaluated used time for features in each of the three categories. While they spent a quite uniform time to implement the "easy" and "medium" features, there were some features which were more complicated to integrate, understand their functionality, or understand the mathematical concepts and algorithms behind them. The outlier "difficult" feature that took around 9 hours corresponded to the `GuessFactorTargeting` including its design to interact with segmentation features, as described in Section 5. The last box-plot in Figure 11 shows the average time spent for all features. It indicates that in average a feature took between one hour to four hours and a half to be extracted or implemented.

7 ROBOCODE SPL AS PLAYGROUND

The Robocode SPL that we make available can be used for different purposes. In this section, we describe some of them that exploit interesting characteristics of the Robocode domain.

Extending it to attributed feature models. The variability resolution in the implemented Robocode SPL is just boolean decisions. Therefore, selected features cannot be parameterized (*e.g.*, through feature attributes). In the current SPL implementation approach, if different feature parameters were really important, different values should have been implemented through different boolean features (*e.g.*, a feature requires to measure if the enemy is far or not, but it is unclear to decide which distance thresholds to use, and different options can represent significantly different behaviors). The Robocode SPL can be extended to support feature parameters.

Automatically finding the optimal bot configuration. Given the combinatorial explosion of possible configurations in an SPL, finding the best configuration for a given context is challenging. Current approaches use to rely on evolutionary algorithms to explore the configuration space [7]. For the Robocode SPL, it is challenging to know which configuration performs the best against a given enemy bot or against a set of bots. The Robocode community has created the RoboRunner framework that enables to launch battles without the user-visible simulation, so the results (*i.e.*, scores) of a battle are automatically obtained. These scores can be used as fitness functions to explore the Robocode SPL configuration space. The optimization can be also multi-criteria, having to balance not only the scores but also other attributes such as code size. Compared to other domains where the fitness function of a configuration is always the same (*e.g.*, cost), a fitness function based on the battle score presents interesting characteristics as the score is not constant each time you run the battle.

Dynamic change of bot behavior. The current Robocode SPL enables to derive bots with a fixed behavior. For example, if you select one type of movement, this movement cannot be changed to other that might be more appropriate for the status of the battle at a given time. The Robocode SPL can be enriched with customizable process models [19] in a way that the configuration is going to be replaced with other configuration at run-time when a set of predefined events are triggered, or when certain battle conditions are satisfied.

8 RELATED WORK

We present the related work from the perspective of extractive SPL adoption as well as from an educational perspective.

Extractive SPL Adoption. The extractive SPL adoption (ESPLA) catalog [15] reports more than 125 case studies. However, only a very few of them consider the analysis of variants that are not created with clone-and-own or through automatic approaches. Only three of them consider variants that are created independently by different development teams. The HomeAway online vacation rental marketplace had to deal with the fusion of several companies of the same domain [12]. 12 variants were implemented independently fitting each company requirements and an SPL wanted to be adopted as part of the fusion. This kind of cases can be hardly managed by tools which are mainly based in structural analysis of the variants, such as BUT4Reuse [16] or ECCO [6]. The BSH induction hobs company [4] also had to deal with the merging of two families counting 112 variants (46 of one family and 66 of another) with cases of features that were implemented independently. The Robocode bots and features are implemented independently by each developer representing a challenging case study for SPL extraction. Also, compared to these industrial cases, the Robocode SPL is made publicly available.

Education on SPL. Some case studies have been prepared to be used in the context of learning and for classroom use. The Arcade Game Maker pedagogical SPL [17] concisely explains the scenario of a fictitious company adopting an SPL approach. Other case studies, such as the Graph Product Line (GPL) [13] or the Expressions Product Lines (EPL) [14] have been presented as SPL examples where understanding the domain is not complex. However, according to a survey and the summaries from the SPL Teaching (SPLTea) series of workshops by Acher et al. [1], there is still need of suitable case studies for educational purposes. The Robocode case study is proposed as a candidate to reduce the lack of well-documented real-world examples and case studies suitable for teaching in the context of reverse engineering and adoption. As commented is Section 1, this context is not as frequently covered as other topics in SPL pedagogical material [1].

9 CONCLUSION

We presented the Robocode case study as a way to analyse an end-to-end extractive process for SPL adoption. The domain analysis resulted in a feature model of considerable size and in a high diversity of feature implementations that were (1) mined from existing assets and (2) re-engineered to match the SPL architecture. We provided time measures to show the distribution of time dedicated to different activities. Our discussions of this academic-scale extraction of an SPL can provide insights for real industrial settings. The case study is also suitable for educational purposes as it enables to gain knowledge and hands-on practice in SPLs through a motivating project.

As further work, we aim to tackle some of the research directions mentioned in Section 7. Also, we plan to get feedback about the domain analysis and implementation from the Robocode community. The Robocode SPL can be used as a reference point to understand, discuss and evolve their domain variability. Also, we hope to attract the attention of the SPL community as the Robocode SPL can be used for very diverse purposes.

ACKNOWLEDGMENTS

This work was partially supported by the ITEA3 15010 REVaMP^2 project: FUI the Île-de-France region and BPI in France. Special thanks to the students Aram, Hacene, Mariène, Morvan, Sara and Wissem for their active involvement in this work.

REFERENCES

[1] Mathieu Acher, Roberto E. Lopez-Herrejon, and Rick Rabiser. 2017. Teaching Software Product Lines: A Snapshot of Current Practices and Challenges. *TOCE* 18, 1 (2017), 2:1–2:31.
[2] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer.
[3] Sven Apel, Christian Kästner, and Christian Lengauer. 2009. FEATUREHOUSE: Language-independent, automated software composition. In *ICSE 2009.* 221–231.
[4] Manuel Ballarín, Raúl Lapeña, and Carlos Cetina. 2016. Leveraging Feature Location to Extract the Clone-and-Own Relationships of a Family of Software Products. In *ICSR 2016*, Vol. 9679. Springer, 215–230.
[5] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS 2013.*
[6] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *ICSME.* IEEE Computer Society, 391–400.
[7] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. 2011. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software* 84, 12 (2011), 2208–2221.
[8] Ken Hartness. 2004. Robocode: using games to teach artificial intelligence. *Journal of Computing Sciences in Colleges* 19, 4 (2004), 287–291.
[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report. Carnegie-Mellon University Soft. Eng. Institute.
[10] Cory Kapser and Michael W. Godfrey. 2008. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering* 13, 6 (2008), 645–692.
[11] Charles W. Krueger. 2001. Easing the Transition to Software Mass Customization. In *Software Product-Family Engineering, 4th International Workshop, PFE 2001 (Lecture Notes in Computer Science)*, Vol. 2290. Springer, 282–293.
[12] Charles W. Krueger, Dale Churchett, and Ross Buhrdorf. 2008. HomeAway's Transition to Software Product Line Practice: Engineering and Business Results in 60 Days. In *SPLC 2008.* 297–306.
[13] Roberto E. Lopez-Herrejon and Don S. Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *Generative and Component-Based Software Engineering, Third International Conference, GCSE 2001, Erfurt, Germany, September 9-13, 2001, Proceedings (Lecture Notes in Computer Science)*, Vol. 2186. Springer, 10–24.
[14] Roberto E. Lopez-Herrejon, Don S. Batory, and William R. Cook. 2005. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP 2005 - Object-Oriented Programming (Lecture Notes in Computer Science)*, Vol. 3586. Springer, 169–194.
[15] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *SPLC.* ACM, 38–41.
[16] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-up adoption of software product lines: a generic and extensible approach. In *SPLC 2015.* ACM, 101–110.
[17] John D. McGregor. 2014. Ten years of the arcade game maker pedagogical product line. In *SPLC 2014, Volume 2.* ACM, 24–25.
[18] Jackie O'Kelly and J. Paul Gibson. 2006. RoboCode & problem-based learning: a non-prescriptive approach to teaching programming. In *SIGCSE ITiCSE 2006.* ACM, 217–221.
[19] Marcello La Rosa, Wil M. P. van der Aalst, Marlon Dumas, and Fredrik Milani. 2017. Business Process Variability Modeling: A Survey. *ACM Comput. Surv.* 50, 1 (2017), 2:1–2:45.
[20] Yehonatan Shichel, Eran Ziserman, and Moshe Sipper. 2005. GP-Robocode: Using Genetic Programming to Evolve Robocode Players. In *EuroGP 2005.* 143–154.
[21] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Sci. Comput. Program.* 79 (2014), 70–85.
[22] Pamela Zave. 2009. Modularity in Distributed Feature Composition. In *Software Requirements and Design: The Work of Michael Jackson.*