An Object-Oriented Framework for Improving Software Reuse on Automated Testing of Mobile Phones

Luiz Kawakami[1], André Knabben[1], Douglas Rechia[2], Denise Bastos[2], Otavio Pereira[2], Ricardo Pereira e Silva[2], Luiz C. V. dos Santos[2]

[1] Motorola - Brasil Test Center
{wlk023,wak023}@motorola.com
[2] Computer Science Department - Federal University of Santa Catarina, Brazil
{rechia,denise,otavio,ricardo,santos}@labsoft.ufsc.br

Abstract. To be cost effective, the decision to automate tests that are usually hand-executed has to rely on a tradeoff between the time consumed to build the automation infrastructure and the time actually saved by the automated tests. Techniques which improve software reuse not only reduce the cost of automation, but the resulting productivity gain speeds up development. Such issues are specially relevant to the software development for mobile phones, where the time-to-market pressure asks for faster design and requires quicker deployment of new products. This paper presents a novel object-oriented framework tailored to support the automation of user-level test cases so as to improve the rate of deployment of mobile phones. Despite inherent test automation limitations, experimental results show that, with automation, the overall testing effort is about three times less than the manual effort, when measured within a one-year interval.

Key words: Software verification, Software reusability, Software metrics.

1 Introduction

Many mobile phone models are released to the market every year with improved or brand new features. Examples of common phone features are messaging (*short messages* – SMS, *multimedia messages* – MMS and E-mail), phone and appointment books, alarm, embedded camera and so on. These functionalities are largely implemented in software. Every feature of each new phone model must be tested prior to its release to the end users. Also, the interaction among features must be checked, so as to ensure their proper integration. Such user-level functional tests are crucial to reduce both customer dissatisfaction and technical assistance costs.

Functional testing relies on checking many use-case scenarios. Each *test case* (TC) is a sequence of steps that performs a specific task or a group of tasks. TCs not only specify the steps, but also the expected results.

Test engineers usually execute TCs manually. A TC may be repeated several times during successive software development life cycles. Manual test execution is both time-consuming and error prone, especially because exact TC reproduction cannot be guaranteed through different phone software versions. Such inadequacy leads to the use of software to automate TC execution. An *automated test case* (ATC) can automatically reproduce the steps that would be performed manually by the test engineer.

This paper presents a novel object-oriented framework tailored to support ATC creation for user-level functional testing of mobile phones. Our *test automation framework*, from now on called TAF, was designed to allow as much ATC reuse as possible across distinct phone models of a given product family. Essentially, TAF allows the automation of functional TCs and the efficient retargeting of pre-existing ATCs to distinct phone models. Such a retargeting within a product family is the very key to achieving productivity gain. Once a TC is automated, it is ready to be executed as many times as needed, through all phone development life cycles, reducing the time-to-market. The main contribution of this paper consists in the analysis of solid experimental results which show the actual impact of software reuse on test automation. The reuse achieved with our framework makes it worthy to be employed in the corporate environment, given certain conditions described later in this paper.

The remainder of this paper is organized as follows: Section 2 reviews related work; the structure of the automation framework is described in Section 3; Section 4 summarizes experimental results and finally our conclusions are drawn in Section 5.

2 Related Work

2.1 Practices in the Corporate Environment

While TC manual execution is still current practice, many companies are widely adopting test automation for unit and regression testing, since they are recurrent during software development life cycles, despite the inherent limitations imposed upon automation (for instance, 50% of the TCs within typical user-level test suites are not suitable for automation).

Although the approaches vary from one company to another, test automation has been progressively adopted at the user-level as a way of reducing the required effort for test execution. At this level, the main approaches employ either in-house developed test ware, such as PTF [1], or third party test systems, such as TestQuest Pro (R) [2].

Motorola relies on in-house test automation infrastructure. TAF, which will be described in Section 3, is the keystone of such infrastructure.

2.2 Related Research Topics

Related approaches on test automation address two basic goals: test case generation, and test execution and analysis.

Test suite generation focuses on finding ways of capturing test cases from code or design specifications. As an example, model-based testing is an approach in which the behavior of the software under test is described by means of formal models (such as Petri nets, state charts and so forth) as a starting point to automatic or semi-automatic TC generation [3] [4] [5] [6]. Other approach relies on algorithms able to create test cases which cover the interaction among different applications in rich-feature communicating systems [7].

The automation of test execution and result analysis aims to produce software artifacts able to execute test suites (automatically generated or not) and to compare the obtained results to the expected ones [8].

Recent work seems to indicate that there is not so far an ultimate solution for software test automation challenges [9]. On the contrary, distinct successful approaches are reported [8], [10], [11], [12], [13].

To assess the economic viability of test automation, a preliminary trade-off analysis [14] should be performed. Since high frequency of invocation is a prerequisite for automating a TC, common pitfalls should be avoided, such as overestimating the required effort for manual execution or underestimating the percentage of tests that are actually suitable for automation [15] [16].

As test automation often consists in producing software to test software, an alternative approach to achieve a better trade-off is to promote software reuse when constructing *testware*. Object-oriented frameworks are reusable software artifacts able to support testware development [17]. JUnit is a well-known example of framework applied to the domain of test development [18].

Since there is a trade-off between generality and effectiveness of reuse, domain-specific frameworks (such as JUnit) are expected to lead to a lower percentage of reuse than application-specific ones. That was the motivation to the development of a novel application-specific framework tailored to mobile phones.

There is lack of evidence in the literature quantifying the impact of software reuse on test automation. That's why the main contribution of this paper is to report the quantitative impact of an application-specific framework on real-life state-of-the-art product deployment.

3 TAF Design Description

TAF is an object-oriented framework tailored to automate functional user-level test-case *execution* for mobile phones. TAF provides the proper infrastructure to automate a test, but this process is essentially manual. In other words, TAF addresses the automation of test *execution*, not the automatic *generation* of tests.

TAF enables reuse by raising the abstraction level so as to make ATCs largely independent of model-specific phone properties. Therefore, it has to rely on a lower-level infrastructure, as described in the following subsection.

TAF was developed by Brasil Test Center (BTC), an R&D network of research institutes under Motorola's leadership[1].

3.1 Low-level Implementation Infrastructure

In order to interface with the phone, TAF relies on a Motorola proprietary artifact, the so-called *phone test framework* (PTF) [1]. PTF provides an application-programming interface (API) that allows the user to simulate events from the phone's input/output behavior, like key pressing and display capture. Since most API methods are encoded at low abstraction levels, PTF leads to test scripts that are hard to read, difficult to maintain and inefficient to port to other phones. However, PTF represents a highly appropriate basis for test automation implementation.

3.2 High-level ATC Encoding

The key to raising the abstraction level is to encapsulate lower-level test input actions (such as sequence of key pressings) and test output analysis (such as checking the phone display contents) into a so-called *utility function* (UF). UFs are primitive entities that hierarchically isolate functionality from implementation, leading to high-level ATCs. An ATC tells "what" to test, but not "how" to perform some input action or output analysis. As a result, UFs must rely on PTF components for actual test implementation.

Fig. 1 shows an example of a high-level ATC using utility functions. This ATC fragment performs the following sequence of steps: first, it takes a picture and stores it as a file (Steps 1 to 3); then, it checks some attributes and deletes the file (Steps 4 to 8). Note that seven UFs are employed: *LaunchApp* (it launches the camera application), *CapturePictureFromCamera* (it takes the picture), *storeMultimediaFileAs* (it stores the picture into the phone file system), *scrollToAndSelectMultimediaFile* (it scrolls through a list and opens a specific multimedia file), *openCurrentFileDetails* (it opens the screen which displays file attributes such as type, size, etc), *verifyAllMultimediaFileDetails* (it checks whether the picture file has the expected properties) and *DeleteFile* (it simply deletes the file from the phone file system).

Although different phones exhibit distinct input/output behavior, a same high-level ATC is applicable to several phone models of a given product family, since they basically implement the same features. Therefore, a TC is automated only once for a product family and the resulting high-level ATC must be retargeted to every distinct phone model within the family. This retargeting process is called *porting*.

TAF was designed to allow efficient porting of high-level ATCs, as will be described in the next subsection.

[1] TAF's initial design and development involved the Computer Science Department of Federal University of Santa Catarina (INE–UFSC) and the Center for Informatics of Federal University of Pernambuco (CIn–UFPE).

```
...
// Step 1: launch Camera application
    navigationTk.launchApp(PhoneApplication.CAMERA);

// Step 2: take the picture
    multimediaTk.capturePictureFromCamera();

// Step 3: store the picture and hold its file name in variable picture
    MultimediaFile picture =
        multimediaTk.storeMultimediaFileAs(MultimediaItem.STORE_ONLY);

// Step 4: take the phone to PICTURES_FILE_LIST screen
    navigationTk.launchApp(PhoneApplication.PICTURES);

// Step 5: open the picture
    multimediaTk.scrollToAndSelectMultimediaFile(picture);

// Step 6: open the file details screen
    multimediaTk.openCurrentFileDetails();

// Step 7: verify picture file attributes
    multimediaTk.verifyAllMultimediaFileDetails(picture);

// Step 8: return to PICTURE_VIEWER
    phoneTk.returnToPreviousScreen();

    multimediaTk.deleteFile(picture, true);
...
```

Fig. 1. An ATC fragment

3.3 TAF Organization

To enable the reuse of ATCs, TAF was designed to overcome the issues that are raised by PTF's low-level APIs, such as creating scripts that are hard to read and difficult to maintain.

Fig. 2 summarizes the organization of TAF in terms of class relations.

The interface *Step* lies at the top of the diagram. It provides a generic method (*execute*) allowing ATCs to invoke the functionality of distinct UFs.

The class *BaseTestCase* stores a collection of objects of type *Step*. It is extended to give rise to a test case (e.g. *Test1*).

On the one hand, the framework relies on key abstract classes that define distinct UF APIs that implement the interface *Step* (e.g. *LaunchApp* and *CapturePictureFromCamera*). They define additional methods to convey UF-specific information (e.g. *setApplication* and *setResolution*).

On the other hand, TAF employs concrete classes to extend UF APIs. UF implementations invoke PTF APIs, thereby enclosing the low-level input/output

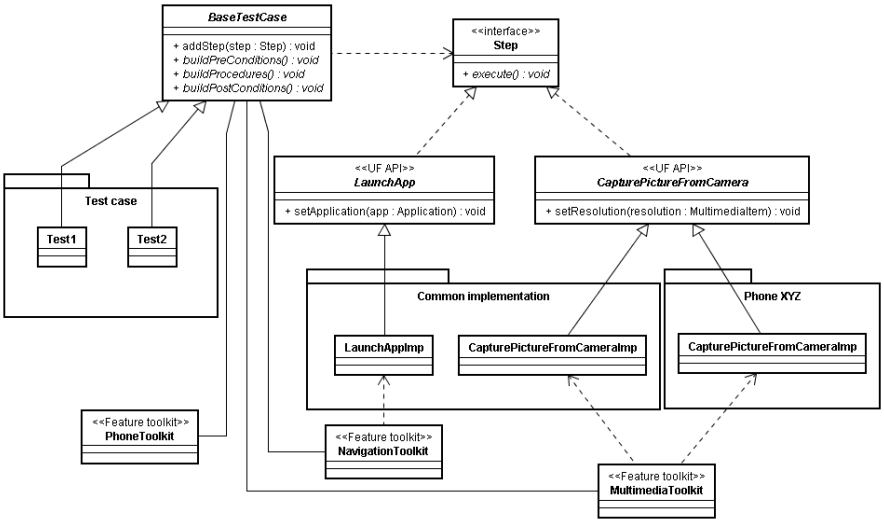

Fig. 2. A TAF class diagram

behavior of a specific phone (e.g. *LaunchAppImp* and *CapturePictureFromCameraImp*). Target-independent and target-dependent classes are organized in distinct packages (e.g. *Common Implementation* and *Phone XYZ*).

To allow proper instantiation of UF implementations for a given phone, TAF relies on the notion of Feature Toolkit (e.g. Phone Toolkit, Navigation Toolkit, Multimedia Toolkit), as illustrated in Fig. 2, at the bottom. Since TAF has potentially more than one implementation for each UF API, this class must know the appropriate UF implementation for the phone under test. This information is encoded within an XML file, which is maintained by TAF developers. Another role of a Feature Toolkit is to add the instantiated UF to the list of test case steps, and to launch their execution. As soon as the step list is created, the test case execution can be started. In brief, an ATC consists of several calls to methods encapsulated within Feature Toolkits. Fig. 3 summarizes the hierarchy of TAF layers from the highest to the lowest level.

3.4 Automating a Test Case with the Aid of TAF

The structure of TAF has facilities to create an ATC from a TC written in natural language and conceived to be manually executed. First, a subclass *BaseTestCase* has to be created as a template for the new ATC. Three of its abstract methods – *buildPreConditions()*, *buildProcedures()* and *buildPostConditions()* – must be overwritten. Such methods define the functional structure of a test: the phone configuration actions required for the test (e.g. date and hour settings, web browser set-up, e-mail accounts, etc.), the actual test steps and post-test clean-up procedures (e.g. the rollback of side effects that could possibly affect

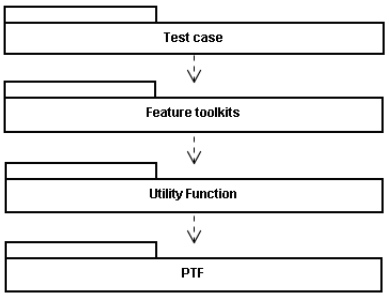

Fig. 3. TAF layer view

further tests). The next step consists in inserting calls to methods of Feature Toolkits. The code to be inserted within the overwritten methods ressembles the one shown in Fig. 1. Once the subclass is created (i.e. the new ATC), a preliminary checking is performed to verify if there are suitable implementations of the required UF APIs for the target-phone. If a proper implementation is found, it will be reused as it is; otherwise, a new one will be created.

3.5 Object-Oriented Framework: a Keystone for Worthy Automation

The object-oriented approach adopted by TAF achieves significant software reuse through three distinct mechanisms: inheritance-based creation of ATCs, reuse of available UFs and porting of pre-existent ATCs to other phones.

Inheritance-Based Creation of ATCs. Remember that the methods mentioned in Section 3.4 provide an interface between the ATC and the UF APIs. Since TAF currently has hundreds of distinct UF APIs available, suitable APIs are very likely to be found for a new test.

Note that, in the worst case, the implementation of a specific UF would require the creation of a subclass of the abstract class in which the UF API is defined, as illustrated in Fig. 2, where the specific implementation *CapturePictureFromCamera* is created when targeting phone XYZ. Note that, even in the worst case (no implementation reuse at all), the process of UF creation is still guided by TAF through the inheritance mechanism.

Reuse of Pre-existent UF Implementations. A new implementation is rarely created from scratch. Sometimes, it may be obtained through the creation of a subclass of a pre-existent UF by partially reusing its code. A non-negligible amount of reuse should be expected in this way, as explained in the following. Consider the classes within the *Common implementation* package (see Fig. 2). Their method *execute()* is a *template* that invokes several *hooks*[19]. Since an

implementation inheriting another implementation must only re-implement the *hook* methods required by a specific phone, all the other methods already encoded in the superclass are reused.

In the best case, an untouched UF implementation is reused. Fortunately, the best case is dominant, as it will be seen in Section 4.1.

Porting ATCs to Other Phones. Given a set of pre-existing ATCs, let's analyze how TAF supports the porting of a test case to another phone. First, in the same way as described in the previous section, it should be checked if every UF in that ATC matches the expected behavior for the phone under porting. If not, a new low-level implementation for this UF must be created. Since UFs are extended only if no compatible UF could be found, TAF maximizes the amount of software reuse. In such a way, the whole code of the ATC is reused for different phone models.

4 Experimental Results

This section presents real-life values collected from BTC's Test Automation Project. Three classes of experiments were performed to assess how effective and sustainable is the impact of TAF on test automation. First, we provide a quantitative breakdown of software reuse induced by the framework (Section 4.1). Second, we quantify the impact of reuse in the process of porting new phones (Section 4.2). Later, we quantify the actual productivity gain by first adding the effort of automating TCs to the effort of executing ATCs and then comparing the overall effort with manual TC execution (Section 4.3).

4.1 Quantifying Reuse upon TAF

Table 1 displays reuse figures measured for a set of 10 phones, each submitted to a same test suite containing 67 TCs. Such suite employs 246 distinct UFs. The second column shows the number of phone-specific UF implementations required to perform the porting to each phone (i.e. the number of new UF implementations created either extending the UF API or extending other UF implementation). The third column shows the number of UFs whose implementations were untouched when reused. The fourth column summarizes the percentage of untouched UFs with respect to the total number of invoked UFs.

The high amount of achieved reuse (84.8% on average) contributes to attenuate the TC automation effort required as a result of the ATC-generation learning curve, as will be discussed in the next section.

4.2 The Impact of Software Reuse

Fig. 4 and Fig. 5 show the evolution of TC automation for two product families during a period of seven months.

Table 1. UF reuse breakdown

Phone ID	# phone-specific UF Imps	# UFs reused	% of reuse
1	56	190	77.24%
2	48	198	80.49%
3	44	202	82.11%
4	44	202	82.11%
5	52	194	78.86%
6	54	192	78.05%
7	58	188	76.42%
8	6	240	97.56%
9	7	239	97.15%
10	5	241	97.97%

Fig. 4 shows the average number of hours required per ATC per developer, normalized to the number of hours required when automation was launched (Month 1). In practice, since the simplest TCs are automated first and the most complex ones later, the TC automation effort increases with time. The minimum effort corresponds to the simplest and shortest TCs. The maximum effort corresponds to the most complex TCs (requiring 1.8 times more effort than in Month 1). On average, the effort is about 1.4 times larger than at the time automation was launched. In brief, Fig. 4 could be seen as a learning curve for ATC generation within typical product families. This is the price to pay to obtain a first ATC, which will be hopefully reused with the help of TAF.

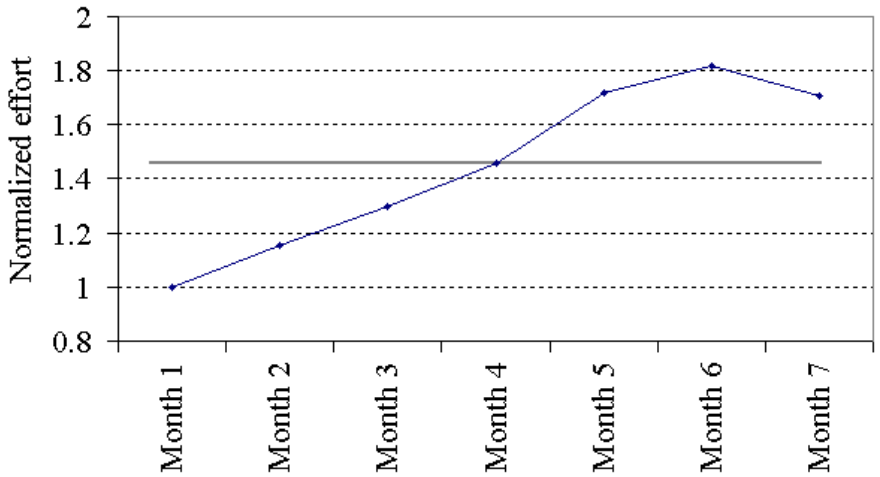

Fig. 4. TC automation effort

Fig. 5 shows the average number of hours required per porting per developer, normalized to the same reference adopted in Fig. 4. Note that the ATC porting effort decreases with time as a consequence of software reuse. The minimum porting effort is approximately 1/4 of the effort required to automate the first TCs. On average, the time to port an ATC to a new phone is about 1/3 of the time to create a new ATC. This is a strong evidence that the TAF architecture effectively enables test reuse.

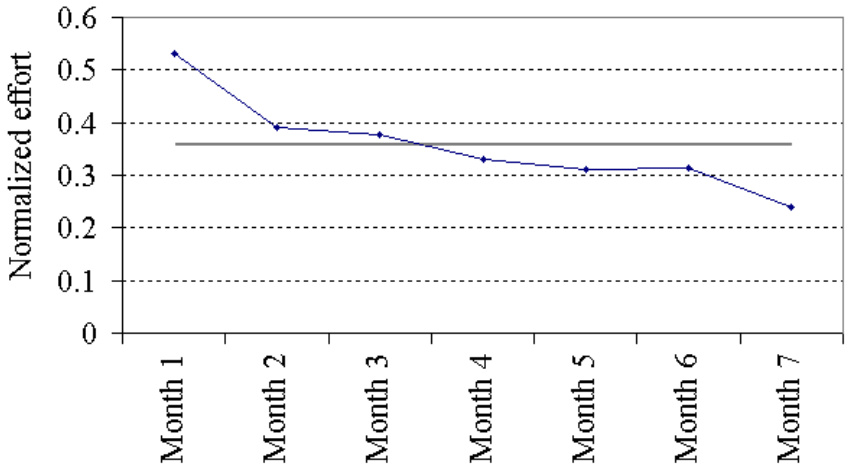

Fig. 5. ATC porting effort

By correlating the average values extracted from Fig. 4 and Fig. 5, we conclude that porting is on average 4 times faster than building an ATC from scratch. This speed up is the very key to achieving productivity gain, as will be shown in the next subsection.

4.3 The Overall Impact of Test Automation

To be worth doing, the overall effort spent in all tasks involved in automation (TC automation, ATC porting, ATC execution and TAF maintenance) must be smaller than executing the same tests manually. In this section, we provide quantitative evidence that, despite the automation limitations at the user level, the adoption of a test framework does pay off.

Experimental Set up. The values summarized in next subsection were measured while testing 15 different phone models belonging to a same product family. Given a phone model, a test suite consisting of 60 TCs (suitable for automation) was selected. First, the TCs were automated gradually, giving rise to ATCs. While an ATC is not completed, its original TC is manually executed instead.

The cumulative effort of testing with the aid of automation was measured along a fourteen-month interval. Since the selected TCs were invoked many times in distinct development life cycles, this procedure captured the overall effort spent with testing.

To assess the impact of automation as compared to purely manual text execution, we performed an estimation of how much time would be spent to run those TCs manually. An estimate of the effort required under purely manual test execution was obtained by multiplying the average manual execution time by the number of TCs that would be invoked for the same test suite during the same period. Such a manual test execution estimate will be used from now on as a reference for effort normalization.

Assessment of Productivity Gain. Fig. 6 depicts the overall testing effort along the monitored period, normalized to the manual test execution effort.

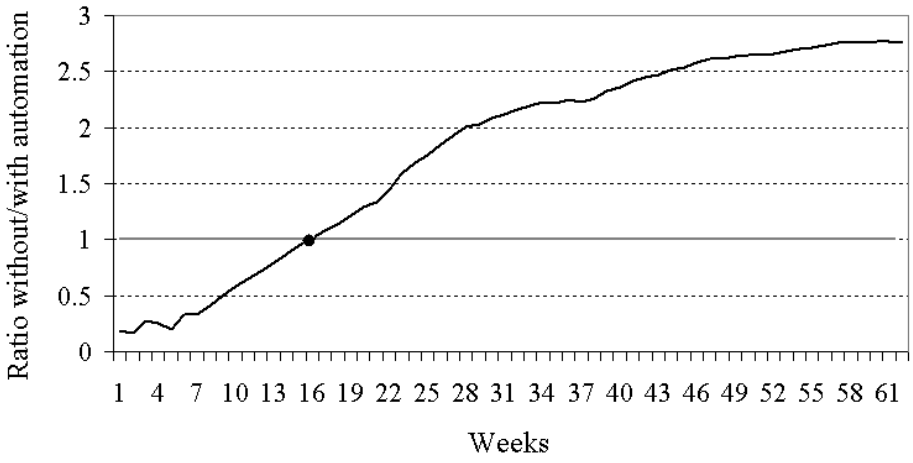

Fig. 6. The impact of automation on effort

Note that the effort to manually execute the test suite for all phones under development within the product family would be 2.8 times larger, if automation was not employed. Therefore, a productivity gain of approximately 3 times was reached within about slightly more than one year. Note also that it took about three months to reach a breakeven point (that is, the time after which automation starts paying off). This indicates that, to deserve automation, TCs must be selected among the highest recurrent ones.

Let's now analyze the impact of automation on each software development cycle. Instead of reporting the cumulative value (as it was shown in Fig. 6), let's now focus only on actual test execution speed-up (i.e. the ratio between the estimated time that the manual test execution team would spend to execute the

whole test suite and the actual time spent to execute the same suite with the aid of automation).

In Fig. 7, dots represent speed-up values obtained within distinct development cycles of various phone models. Note that, the speed-up is 2 on average. This means that, with automation, the whole test suite is executed twice as fast as compared to purely manual test execution. In other words, assuming a constant number of test engineers, manual test execution takes twice as much time to deliver test results.

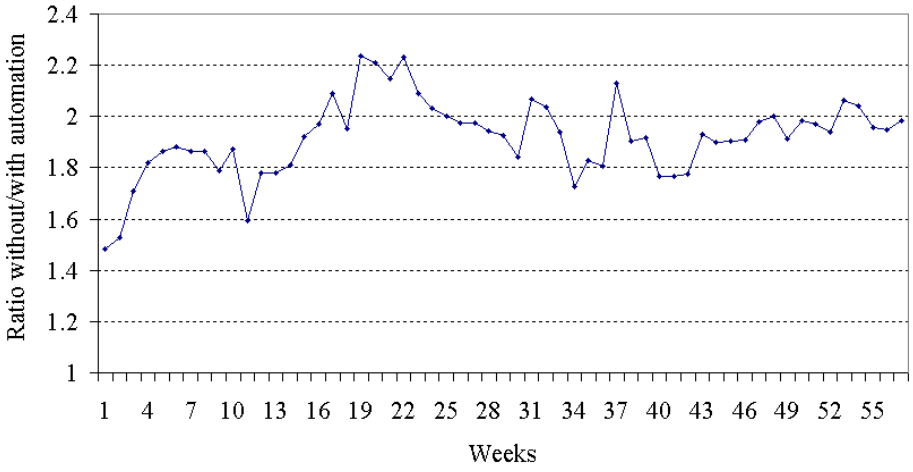

Fig. 7. Test execution speed-up with automation

5 Conclusions and Future Work

We reported an object-oriented framework granting significant test productivity gain by means of software reuse.

As opposed to most reported approaches, we present abundant quantitative evidence of the impact of test automation on real-life state-of-the-art mobile phones. Experimental results indicate that a productivity gain of three times can be achieved in about one year. To reach such a gain, it was shown that the porting of a pre-existing ATC should be around four times faster than automating an equivalent TC from scratch.

As future work, our framework will be extended to support other mobile phone platforms. We also intend to employ Aspect-Oriented Programming and Formal Verification so as to detect possible flaws in the test software.

References

1. I. Esipchuk and D. Validov, "PTF-based Test Automation for JAVA Applications on Mobile Phones", Proc. IEEE 10th International Symposium on Consumer Electronics (ISCE), 2006, pp. 1–3.
2. Test Quest, "Test Quest Pro", available at http://www.testquest.com, 2006.
3. A. Pretschner et al., "One Evaluation of Model-Based Testing and its Automation", Proc. International Conference on Software Engineering, 2005, pp. 392–401.
4. S. R. Dalal et al., "Model-Based Testing in Practice", Proc. International Conference on Software Engineering, 1999, pp. 1–6.
5. J. Bredereke, B. Schlingloff. "An automated, Flexible Testing Environment for UMTS", Proc. 14th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems, 2002, pp. 79–94.
6. T. Heikkilä, P. Tenno, J. Väänänen. "Testing Automation with Computer Aided Test Case Generation", Proc. 14th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems, 2002, pp. 209–216.
7. C. Chi, R. Hao. "Test Generation for Interaction Detection in Feature-Rich Communication Systems", Proc. 17th IFIP TC6/WG 6.1 International Conference, TestCom, 2005, pp. 242–257.
8. Tkachuk, O. and Rajan, "Application of automated environment generation to commercial software". Proc. International Symposium on Software Testing and Analysis, 2006, pp. 203–214.
9. H. Zhu, et al. "The first international workshop on automation of software test". Proc. International Conference on Software Engineering, 2006, pp. 1028–1029.
10. L. Gallagher and J. Offutt, "Automatically Testing Interacting Software Components", Proc. International Workshop on Automation of Software Test, 2006, pp. 57–63.
11. J. C. Okika et al., "Developing a TTCN3 Test Harness for Legacy Software", Proc. International Workshop on Automation of Software Test, 2006, pp. 104–110.
12. S. Xia et al., "Automated Test Generation for Engineering Applications", Proc. International Conference on Automated Software Engineering, 2005, pp. 283–286
13. S. Kansomkeat and W. Rivepiboon, "Automated-Generating Test Case Using UML Statechart Diagrams", Proc. of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology, 2003, pp. 296–300.
14. R. Ramler and K. Wolfmaier, "Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost", Proc. International Workshop on Automation of Software Test, 2006, pp. 85–91.
15. S. Berner et al, "Observations and Lessons Learned from Automated Testing". Proc. International Conference on Software Engineering, 2005, pp. 571–579.
16. J. Oliveira et al., "Test Automation Viability Analysis Method", Proc. VII IEEE Latin-American Test WorkShop (LATW2006), 2006.
17. M. E. Fayad et al., "Building Application Frameworks: Object-Oriented Foundations of Framework Design", Prentice Hall, 1999.
18. E. Gamma and K. Beck, JUnit specification, available at http://www.junit.org, 2006.
19. E. Gamma et al. "Design patterns: elements of reusable object-oriented software". Reading: Addison-Wesly, 1994.