UDC 004:37

Yevgueny Kondratyev
Independent Software Developer, Dnipro, Ukraine

TECHNICAL OPTIMIZATION OF CROSS-PLATFORM SOFTWARE DEVELOPMENT PROCESS QUALITY AND USABILITY OF 3RD-PARTY TOOLS

DOI: 10.14308/ite000595

The article exposes developer's point of view on minimizing creation, upgrade, post-release problem solving time for applications and components, targeted to multiple operating systems, while keeping high end product quality and computational performance.

Non-uniformity of analogous tools and components, available on different platforms, causes strong impact on developer's productivity. In part., differences in 3rd-party component interfaces, versions, quality of distinct functions, cause frequent switching developer's attention on issues not connected (in principle) with the target project.

While loss of development performance because of attention specifics is more subjective value, at least physical time spent on tools/components misbehavior compensation and normal tools configuring is measurable.

So, the main thesis verified is whether it's possible to increase continuity of the development process by technical improvements only, and by which value.

In addition, a novel experimental tool for interactive code execution is described, allowing for deep changes in the working program without its restart. Question under research: minimizing durations of programming-build-test-correct loop and small code parts runs, in part., improving the debugging workflow for the account of combining the interactive editor and the debugger.

Keywords: *cross-platform, programming, usability, optimization*

Impulse for the below research emerged from many-year practice in programming C++-based programs and components in different areas (science, economics, graphics tools, system tools, specialized plugins).

All such projects require at least C++ compiler and system libraries, available under target operating system (OS). Almost none of them, also, avoids using 3rd-party tools (like text editor with syntax highlight and code navigation) and components (application libraries, additional system libraries, binary agents, network services and so one).

Modern software development industry tends to technology racing, when analogous software products are issued in very short period of time, relying on further issuing new versions containing additions, improvements, and bugfixes.

In practice, frequent version change is not optimal for quality growth, and also, from another side, causes additional delays on adapting new versions, detecting their incompatibilities and newly inserted issues (both consciously and unconsciously), as well as time spent on installation and configuring for personal needs (in case of experienced developer, such needs are very much specific and cannot be outsourced to low-cost specialists).

In hypothetic situation, such factors could be compensated from both developer sides (tools/components suppliers and "users"), by several means.

1. (tool developer) Keeping certain subset of features unchanged:

- Groups of interface methods, dedicated to one logical task, should not be altered. In particular, changing method name in a component causes all developers, using the component, face that change and memorize the new name in addition (!) to the old name, which remains actual for rather long time, namely, until any previous commits of the client project lose their actuality. Changing the order and/or default values of method arguments cause worse problem, in addition to the above. Project code may pass compilation successfully and then malfunction, because the implied values are changed. In the modern iterative approach, each 3rd party component version change in such fashion breaks not only work plan and time estimate, but causes unpredicted behavior of the released program.
- Formats of configuration and log files should be defined once to successfully reflect any future needs of the program, or at least, should be backward compatible. The key here is that logs and configurations may be used by the client project developer in a way that could not be imagined by the 3rd party component developers. This is common case with specialized libraries, both open- and closed-source. For closed-source components, programmatic configuring and log reading are often the only means to adapt the library to particular needs. Open-source code, potentially, can be corrected as necessary, but this is often not affordable, because 3rd party code changes must be re-made and retested each time the open-source library version is updated. This would result in continuous time expense.
- Identifiers renaming as part of code refactoring (itself a great method) is very much favored practice in the modern programming. Regrettably, low and average performance programmers do not take into account the fact that human eye has great capability to stick to physical image, namely, to remember and quickly catch words and phrases as whole, provided that their look (font, color, size) varies within certain narrow range. This, conjoint with fixed screen layout (fixed text windows size, fixed sequences of menu items, fixed icon images) allows for very fast reflex-level working on the code.

3. (tool developer, client project developer) Pre-configuring hotkeys and visual layouts of developer tools to certain common minimum, without excessive elements. The "new GUI look to improve user performance" is most often the utopia, because the developer tool developers have no adequate means to estimate what parts of the GUI are critical for user's performance, including not only users with "average" needs, but also users with highly specialized needs, in part., ones who actively write programs for over-application control, to automate whole workflow, consisting of many tools, developed by many companies. In all the modern operating systems, console programs, fortunately, tend to honor specific user needs, by keeping tens or hundreds switches and parameters working unchanged across versions. GUI applications, in contrary, tend to change layouts, formats and behavior with each version, which is most often very unlikely for advanced developer's performance.
4. (tool developer, client project developer) Active improving tool response time in frequent operations (e.g. full project recompilation). Together with fixed names, layouts, reflex-based working, and workflow automation (as mentioned above), responsive tools push the experienced developer's performance to its physiological boundary, because actions on the computer system are performed more based on the integral image, reflected in the human mind, than as the result of looking at the screen. Computer screen here is only secondary tool for bi-directional synchronizing the imagined system state with its physical state (RAM, HDD, network).
5. (tool developer) Active improving component bottleneck functions performance.
6. (tool developer) Rarer new version issuing, while supporting the existing versions that were most appreciated by developers community, by means of architecture- and interface-saving patches.

In the present research, time-study is done for 3 kinds of project activity: programming during new development, feature adds to already released project, issues solving on the new version on the already released project. Developer tools/components used were mixed approx. in the following fashion: 50% versions and UI configuration are kept unchanged during for more than 3 years; 50% are installed anew to match modern OS versions and end product compatibility requirements, and manually pre-configured as close as possible to the older versions.

From both the time-study and previous development experience, only part of activities may be identified as "normal" (i.e. productive, planned and predictable).

Another part is, in general, facing expectations breaking on different levels (user, programmer) and immediate effort to fix the problem, which causes attention switching.

Essential part (not complete list, of course) of "normal" activities:

- (once or rare) creating requirements specification, elaborating it between developers and their customer,
- (once or rare) selecting technologies (internet search, testing, testing existing solutions similar in functionality),
- (once or rare) creating distribution and update packages,
- (once or rare) creating OS and product installations (for testing, for end users),
- (once or rare) installing, configuring developer OS and tools,
- (frequent) thinking,
- (frequent) new code writing/compiling/testing/correcting, old code rewriting,
- (frequent) 3rd-party code adapting and rewriting,
- (frequent) formal test runs, debugging, issue reproduction, correction,
- (frequent) documenting,
- (frequent) commits, backing up, diary, todo, issue tracking, communication tracking (manual cataloguing mail, chat, voice, desktop sharing records for efficient access),
- (frequent) internet search/reading: technology or product specifications, discussions for issue solving,
- (frequent) planned communication between project participants.

Main categories of non-productive activities:

- (frequent) technical operations (OS boot, network connection, manual file management, programs open/close/tuning etc.),
- (time-to-time) satisfying interest in items, unrelated to the current project, when occasionally found in the network during normal workflow; ongoing self-education process,
- (frequent) passively spent build time,
- (time-to-time) compensation of tools and components misbehavior (multiple: UI issues, code incompatibility, inconsistency with descriptions, implicit limitations, unknown bugs and issues, hardware and OS problems),
- (time-to-time) unplanned activity (communication between project participants and other people, breaks during work time etc.)

The more detailed list of factors, influencing attention weakening or switching.

Remark. Very possibly, part of problems due to irresponsible development by leading software companies and less professional developers is larger than part going from physical or technical limitations of system used and chosen tool/component architecture.

So, some partial recommendations are added to keep attention tight on different sides.

- OS response time during boot and normal operation. This can be greatly reduced by combining the following:

1. Using SSD instead of HDD.
2. Increasing RAM volume.
3. Disabling unnecessary OS services, scheduled tasks, automatic recovery etc.
4. Disabling unnecessary internet connections on the firewall level.

5. Disabling GUI visual effects.

- Spontaneous OS self-activity slowing down normal program response. Even with the above optimizations, it does not disappear completely, and requires individual research for each case.
- Network/browser response time. Browser response time is reduced by individual browser settings, related to automatic network services. Network response time bottlenecks removal may have different causes, requiring individual approach.
- OS and program GUI issues breaking normal workflow. While problems that occur regularly may be solved as early as possible (experimentally and/or using online issue discussions), spontaneous issues (driver faults, system hanging, high CPU consumption by certain programs for unknown reasons etc.) are unavoidable.
- Program GUI constantly changing controls, informational areas, list items position on the screen (by design). This is discussed above.
- Occasional system crashes. Due to multitude of ongoing optimizations and overall instability, whole image of operating system and programs should be backed up on regular basis. Practice shows that no modern system can be regarded as completely stable, especially when fully equipped with 3rd-party software according to developers needs. To benefit from backups, each used program must be researched and configured individually in the way that all its variable data (files, registry entries) is kept out of the system image. Variable data should be backed up separately.
- Extra full rebuilds due to incremental build misbehavior. In some cases, may be solved by multi-core and distributed building. Generally, the choice for each build is intuitive, based on the level of changes (header files, declarations, macro definitions vs. function bodies only).
- New or new-version 3rd-party component requires special tools/environment and/or correction to build correctly. In this situation, to build the component only is generally not enough. The new tool has to be built into the current workflow and automated along with the currently used tools.
- New or new-version 3rd-party component requires too much redesigning to be integrated, so that alternate solution search attempt is necessary. Rewriting or alternative implementation may be the choice if estimated time expenses on development and support for the new code during whole lifetime are less than that of redesigning each new version of the initial component.
- 3rd-party component appears faulty or bad-coded in the middle of the development phase. The above re-implementation is the only choice to avoid after-release problems.
- Non-optimal 3rd-party algorithm may lead to application performance loss. This should be tested early, on prototype, if possible. Unfortunately, re-implementing fine algorithms is not always affordable.
- Generalization in any set of functions may lead to application performance and feature loss. This also should be tested early. Note that keeping specialized versions of functions increases project size and complexity.
- Good component may lose performance if misused or non-optimally used on the client side. The developer has to try controlling the way of using his component or tool by his "clients", themselves developers. Possibly, some modeling client's behavior may be done to find the most weighing problems, before the component is released. One of the good practices with general-purpose components is releasing the component or its part as open-source project. This works well even with commercial components, when they are part of some major project. Testing the component by multiple specialists in unpredictable conditions allows for adapting its features to common expectations.
- Good tool or component may become unusable in conditions of high system or task load. Stress-testing should be a must for responsible projects, regardless of time spent on it.

Automatic test cases should be written for each feature, conditional branch, and parameters set, including unused branches and edge cases. In case of large projects, where complete testing is not possible, at least all main functions must be tested before closing the project [2]. Before implementing any new feature, longer than 1 man-month, a prototype should be created and tested. Some authors [1] recommend to throw out the first version of any system.

- Good tool or component may become unusable if it's based on shorter lifetime component. Avoid technologies, whose owners too frequently change terms of their product use and/or drop support for well-working products.

Note. Some of the described factors are touched by [3], a huge research concentrating on human factor rather than technical means.

Time-study results for randomly chosen 10 hours

1. Normal development activity.
   Average time: 66%.
   Max. time in distinct session: 95%.
2. Technical operations.
   Average time: 7%.
   Max. time in distinct session: 29%.
3. Passively spent time during system being busy (build, connect, file copy, calculations).
   Average time: 0.6%.
   Max. time in distinct session: 1.1%.
4. Compensation of tools and components misbehavior.
   Average time: 18%.
   Max. time in distinct session: 28%.
5. Unplanned activity.
   Average time: 9%.
   Max. time in distinct session: 20%.

Interpretation

First of all, max. development activity is nearing 100%. This shows theoretical boundary of productivity.

Technical operations are mostly unavoidable. Some small part of time may be saved by configuring tools uniformly in all OSes as close to physical reflexes of the developer. In part, the most frequently used tools (like text editor) must be chosen by minimal startup and response time.

Passive time due to system load may be larger in certain projects. Frequency of rebuilds depends on task being solved and developer proficiency. If avg. number of rebuilds in medium-size project is less than 10-20 per day, technical speed-up for write/build/test/correct loop is rarely necessary. Still, the loop optimization is valuable because of sideway tasks, prototyping, stress and stability testing. One of the approaches, currently under research, is described in the below section.

Unplanned activity cannot be decreased in formal way, so its part remains as is. This value roughly coincides with tests performed by other researchers.

Tools and components misbehavior lays the heaviest stress on developer's attention. Curiously, time part of facing/solving this kind of problem is almost unchanged from session to session. Also, it's noticed from the experience, that during each of more than 90% sessions, at least one such problem emerges.

It must be noted that in different development niches and sectors the above values may noticeably differ, also they depend on developer proficiency level. The present research does not intend to cover all cases (it's not possible), but highlight several harassing problems of modern software development culture.

Experimental tool for interactive coding in C++

For the programmer, using C++ language in the multitude of various projects, it is often necessary to quickly test small scattered fragments of a program. As a rule, operating system,

build environment, compiler version and other technical conditions are rigidly specified in each particular project, and noticeably differ between projects.

Designing experience and level of problems, solved by specialist, are constantly growing, and the interest in operating system nuances, language features and their implementation in the compilers, is growing as well. Integrating source code and third-party components causes endless research for increasing reliability and avoiding technical flaws.

Existing language interpreters are only partially applicable to such tasks. Existing debuggers allow for working on the source level and link with binary representation, relying on the particular compiler specifics, but many of them have obvious problems with manual editing complex objects on the fly (i.e. while standing on the breakpoint) and too long response time (for specialist - not an average user).

During years, these inconveniences become distressing. Although, it's well known how difficult it is to create a compiler alone from scratch, keeping within the thousand-paged standard. Each IDE, interpreter, debugger are also huge projects.

So, a flexible, simple and universal tool, raising the convenience of compiler-oriented language to the level of interpreted languages, seems hardly implementable, esp. by personal effort in open-source fashion. Major obstacles: wide range of needs and giant time expenses.

Nonetheless, time is going tirelessly, and one day there comes an idea.

1. Interactive C++ editor (including, later, commands for system shell etc.).
   1.1. Instructions input via arbitrary text editor, initially - notepad.
   1.2. Automatic compiling and execution after specific keypress.
   1.3. Both the source code and run-time console output are automatically put into the second window of the text editor. The programmer may freely edit, copy-paste etc. any part of any text. The interactive editor minimizes the cycle of code writing and debugging, also making possible efficient problem solving in areas, traditionally serviced by interpreted languages.
2. The monitor program, implementing the above operations, is also responsible for keeping the run-time context (global variables) in RAM during session.
3. Each new portion of instructions, when compiled and run, sees all existing declarations and global variables. The interactive code preprocessor must distinguish between the following kinds of blocks: directives (#include, using...); declarations (structures, functions...); global variables declarations; statements for immediate execution. To take into account editors with code navigation, methods of separating instructions and including the existing declarations must be configurable. Still, for simpler prototyping, the initial version uses specific character sequences (`, ``, `1, `2) to distinguish between blocks.
4. When any window of the text editor is manually closed, the monitor program automatically calls destructors for all global variables, frees memory, unloads dynamic modules, closes all additional windows.

An example of interactive code follows.

```
`1#include <ctime>
`2double max_delta(const vector<double>& qq)
{
 if (qq.size() < 2) { return 0.; }
 double dt0 = qq[1] - qq[0];
 for (unsigned int i = 2; i < qq.size(); ++i) { double dt = qq[i] - qq[i-1]; if (dt > dt0) { dt0 = dt; } }
 return dt0;
}
`vector<double> qq;
``int t1 = clock(); qq.push_back(t1 / 1000.);
```

```
while (qq.size() < 10) { while (true) { int t2 = clock(); if (t2 != t1) { t1 = t2; break; } } qq.push_back(t1 / 1000.); }
cout<<"Timer resolution, s: "<<max_delta(qq)<<flush
```

It's easily noticeable that the code is not pure C++. It consists of several sections, separated by special character sequences. They can be input together or separately. Anyway, the final autogenerated session code will be functionally the same:

```
#include <windows.h>
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;
#include <ctime>

struct ___session
{
 void ___f1()
 {
  cout<<"Hello, World!\n";
 }

 double max_delta(const vector<double>& qq)
 {
  if (qq.size() < 2) { return 0.; }
  double dt0 = qq[1] - qq[0];
  for (unsigned int i = 2; i < qq.size(); ++i) { double dt = qq[i] - qq[i-1]; if (dt > dt0) { dt0 = dt; } }
  return dt0;
 }
 struct ___d2 { long long ___sep; vector<double> qq;  }; long long ___sep2;
 vector<double> qq;
 void ___f2()
 {
  int t1 = clock(); qq.push_back(t1 / 1000.);
  while (qq.size() < 10) { while (true) { int t2 = clock(); if (t2 != t1) { t1 = t2; break; } } qq.push_back(t1 / 1000.); }
  cout<<max_delta(qq)<<flush;
 }
};

#define __ICPP_DLLEXPORT extern "C" __declspec(dllexport)
__ICPP_DLLEXPORT void __icpp_addinit(void* pd) { new (&((___session*)pd)->___sep2) ___session::___d2(); }
__ICPP_DLLEXPORT void __icpp_exec(void* pd) { ((___session*)pd)->___f2(); }
__ICPP_DLLEXPORT void __icpp_destroy(void* pd) { ((___session*)pd)->~___session(); }
```

This is pure C++ code, suitable for compiling a DLL. The interactive editor ensures building the DLL, loading it into its own address space, and calling __icpp_exec. Possible console output is redirected into file and displayed to the user (programmer) when the function exits.

Note. OS choice for the first experimental implementation of interactive editor is Windows. In POSIX system, something analogous is easily achievable. The most important is keeping simplicity and small volume of source code (the working program does not exceed 1500 lines, written during 3 days), so that any interested programmer could adapt it to personal needs.

Factual interactive C++ host algorithm is rather intricate. Still, main sequence, without details, is simple:

1. Allocating and holding an area of dynamic memory for session variables (plus certain amount reserved).

   Main loop:
2. (*) Watching for Ctrl+Enter keypress in the text editor window ("window #1"). If the window has been closed, go to step 12.
3. (*) Get the text from window #1.
4. Construct the source code of the session.
5. Compile a DLL from the session code.
6. (*) On compilation error, print results (console text) into the text output window ("window #2").

   On compilation success:
7. Print session code into window #2.
8. Clear window #1.
9. Load new DLL, call __icpp_addinit.
10. Output step 9 results into window #2.

    On step 9 success:
11. Call __icpp_exec. Output results into window #2. Go to step 2.

    Completion:
12. Call __icpp_destroy of all loaded DLLs, in reverse order. Free the memory block left after destroying session variables.
13. Close windows #1 and #2.
14. Exit the monitor program.

Note. Non-trivial functions are asterisk-marked (first occurrence). Their implementation depends on programmer's needs, and also on the programs used as text windows. When porting the monitor program into different OSes, these functions may also be implemented differently.

Interactive C++ implementation properties and benefits

For interactive C++ editor, there are many obvious applications, the first ones are speeding up the coding-compiling-debugging cycle, and modifying variables in the working program. There may be several scenarios of interaction between the monitor program and the context of target program, for example, injecting a series of hook DLLs into the working program instead of loading session DLLs into the monitor program context, reading and modifying variables by addresses, taken from the debugger, sending/receiving messages and synchronized access to variables in the multithreaded context, manipulating threads through system API calls etc. Implementing such functions for individual needs does not exceed several days.

With default configuration, factual program startup time is about 0.2 s. Response time for executing a simple C++ instruction is about 1 s, where 80% is spent by the compiler.

The first tests of the interactive editor were conducted on rather complex project - the library for synchronous capturing desktop images sequence with windows filtering. Testing showed what's necessary to add for getting everyday tool for real-time debugging multithreaded applications. Major changes: configuration file format, text windows and clipboard control algorithm. The tested library also required certain modifications, caused stability improvement on loading and unloading (static variables are moved into the single dynamic object, inherited during debug by all loaded DLLs; addresses of loop procedures, executed in the dedicated threads during screen capture, are dynamically updated on loading each new instance (version) of the session DLL).

The described approach allows for debugging the program without breaking or restarting, in part., modifying algorithms, arbitrarily modifying and printing values of variables. For example, on the below screenshot you can see a moment of applying changes in the code line, printing certain debug info. Initial line:

cerr << "T " << GetTickCount() << " ifr1 " << ifr1 << " t_est " << t_est_prev << "\n";

New line:

cerr << "T " << GetTickCount() << " ifr1 " << ifr1 << "\n";

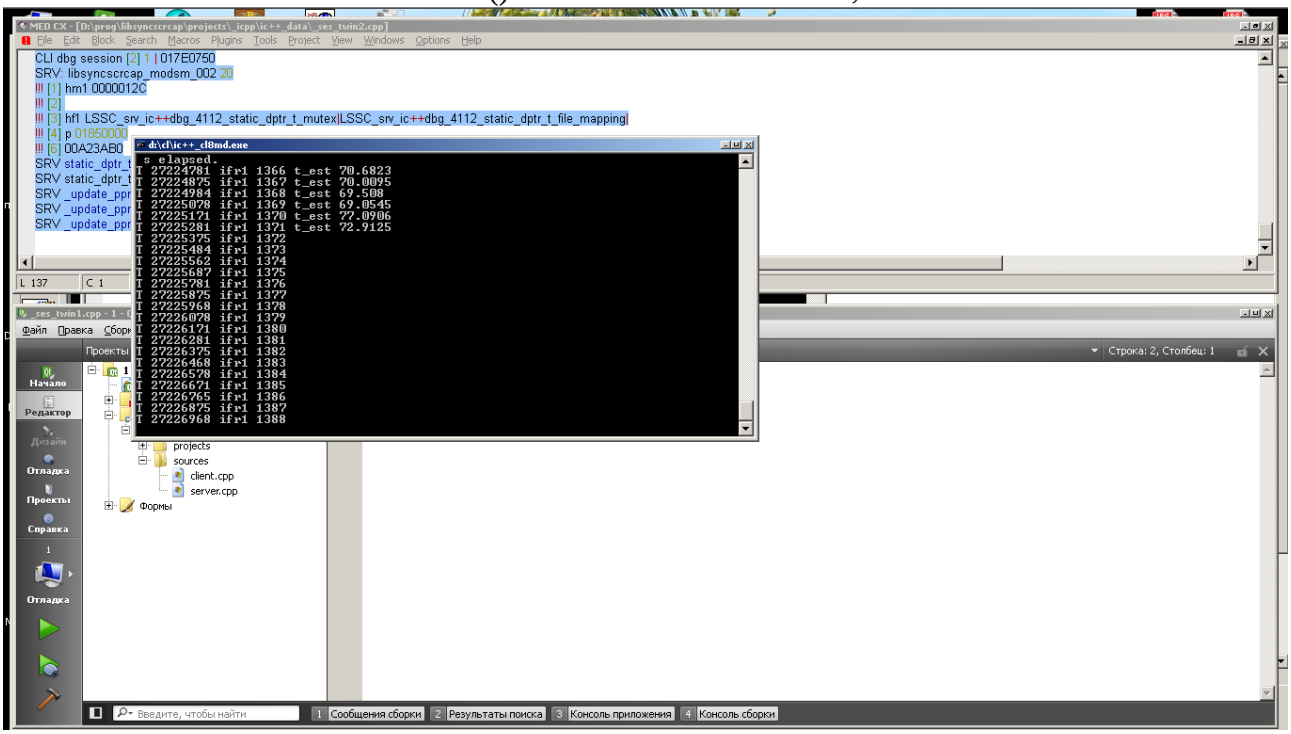

Fig. 1. A screenshot, taken at the moment of interactive applying changes in the C++ code line.

The interactive editor supports multiple text editors and compilers, tunable via configuration file. Here, it is necessary to make several notes.

1. From the point of view of the target project, supporting several compilers increases code quality.
2. Not all builds and combinations work ideally in the context of interactive DLL host, presumably due to underlying libraries implicit use of inner static variables. Depending on the compiler and OS version, either static or dynamic builds may perform better (i.e. without casual faults). Console output redirection may behave incorrectly, if build parameters of the monitor program differ from that of the target project (session DLLs), because the monitor program is at the same time the session DLL host.
3. The debugging itself is the most pleasant and effective with the compiler, having the least time of getting the binary module (session DLL) - the major component of response time for user sending a portion of interactive code. Here, the situation with "what compiler is better" is dramatically different from multiple discussions available in the Internet. For example, among tested compilers, "the best" is cl.exe, released with VS 2008. It's response time when recompiling 5000 lines of the target project (the capture library) is 2..4 s. For comparison, 2015'th g++.exe spends 6..14 s, icl.exe - about 15 s, which noticeably breaks interactivity of the debug workflow.

Obviously, combining the described approach with traditional debug workflow may require certain debugger tuning. In return, it gives the flexibility unreachable with interpreted languages in complex projects: simple and fast access to all levels of code and data, down to physical addresses, in the environment (compiler, IDE, OS) exactly matching with customer

requirements. Porting the interactive editor into different OSes should not be difficult because its source code is very small and uses only standard system APIs.

Resume

In the scope of present research, it must be concluded that developers, having more than average responsibility for their projects, are very limited in 3rd-party means to base on, especially when the project is targeted to two or more platforms.

This should be taken into account by all interested parties, because the long-term physical limit of processing unit technology is already reached. Modern OSes, IDEs, component bundles' size, responsiveness, performance, are far from optimal, yet farther from theoretical boundary. Multi-core calculations solve only part of the problem. The most essential bottlenecks in programs remain sequential. It should be noted by many leading developers and project managers, that the nature of programs is qualitative, not quantitative. Economical approach (in part., "product", "industry", "productivity", "profit" terms) is almost fully valid for physical manufacturing and more-less for agriculture. In software development, it's only simulation, valid during relatively short period of time. Software "product" may live long, only being and remaining exceptional in some of its properties, for example, near-theoretical response time for everyday use tool, or, in computational area, 10x calculations speed in comparison with analogous products.

Product uniqueness is important, but not the only sign of projects, requiring technical optimization. To benefit from that, project participants must have certain level of software development culture, based on past experience and personal self-development efforts. This implies management's ability to hire and keep exceptionally gifted people [2].

On the educational level, talented students should be assisted by the older, experienced, specialists, in removing professional system usage bottlenecks. As practice shows, many of the students, capable of solving algorithmic tasks, spend too many time and efforts when misusing development tools. Also, on another level, insufficient knowledge of system programming and operating system internal operation, leads to misusing good 3rd party products. This sometimes causes 10x loss of end product performance, and introducing high-level bugs.

Currently, a specific research is conducted on the described problem. Also, as described, one of C++-specific problems (tendency to longer build time and slower automatic testing when the project grows) is attacked by developing an experimental tool for interactive code execution. Interactive C++ code execution allows making tens to hundreds varying tests per hour on the working program, without increasing complexity of the developer's toolset.

REFERENCES

1. Frederick P. Brooks. The Mythical Man-Month: Essays on Software Engineering. 1975, 1995.
2. Edward Sullivan. Under Pressure and On Time. 2001.
3. Andre N. Meyer, Thomas Fritz, Gail C. Murphy, Thomas Zimmermann. Software Developers' Perceptions of Productivity. 2014. URL: http://research.microsoft.com/pubs/228971/meyer-fse-2014.pdf

Стаття надійшла до редакції 19.03.16

Кондратьєв Є. В.
Дніпро, Україна

ТЕХНІЧНА ОПТИМІЗАЦИЯ ПРОЦЕСУ РОЗРОБКИ КРОС-ПЛАТФОРМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ТА ЗРУЧНОСТІ ВИКОРИСТАННЯ ІНСТРУМЕНТІВ

Стаття розкриває точку зору розробника програмного забезпечення на зведення до мінімуму часу створення, оновлення, коригування програм і компонентів, призначених для декількох операційних систем, при збереженні високої якості кінцевого продукту і продуктивності обчислень.

Неоднаковість аналогічних інструментів і компонентів, доступних на різних платформах, має сильний вплив на продуктивність розробника. Зокрема, відмінності в інтерфейсах компонентів третіх сторін, версіях, якості окремих функцій, викликають часті переключення уваги розробника на проблеми, не пов'язані (принципово) з цільовим проектом.

У той час як оцінка величини втрати продуктивності розробки через особливості уваги має більш суб'єктивний характер, принаймні фізичний час, витрачений на компенсацію неправильної поведінки інструментів та компонентів, може бути вимірено.

Таким чином, основна теза, що перевіряється – чи можливе збільшення безперервності та продуктивності процесу розробки за рахунок тільки технічних удосконалень, і на яку величину.

Додатково, розглянуто новий, експериментальний інструмент для інтерактивного програмування. Інструмент дозволяє вносити глибокі зміни у програму в процесі її роботи, без перезапуску. Досліджуване питання: мінімізація тривалості циклу програмування-компіляція-тестування-корекція та перевірки окремих невеликих частин коду, зокрема, удосконалення робочого процесу налагодження за рахунок сумісного використання інтерактивного редактора та налагоджувача.

Ключові слова: крос-платформний, програмування, зручність використання, оптимізація

Кондратьев Е. В.
Днепр, Украина

ТЕХНИЧЕСКАЯ ОПТИМИЗАЦИЯ ПРОЦЕССА РАЗРАБОТКИ КРОСС-ПЛАТФОРМЕННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ КАЧЕСТВА И УДОБСТВА ИСПОЛЬЗОВАНИЯ ИНСТРУМЕНТОВ

Статья раскрывает точку зрения разработчика на сведение к минимуму времени создания, обновления, пост-релизной коррекции приложений и компонентов, предназначенных для нескольких операционных систем, при сохранении высокого качества конечного продукта и производительности вычислений.

Неодинаковость аналогичных инструментов и компонентов, доступных на различных платформах, имеет большое влияние на производительность разработчика. В частности, различия в интерфейсах компонентов третьих сторон, версиях, качестве отдельных функций, вызывают частое переключение внимания разработчика на проблемы, не связанные (принципиально) с целевым проектом.

В то время как количественная оценка потери производительности разработки из-за особенностей внимания имеет скорее субъективный характер, по крайней мере физическое время, затраченное на компенсацию неправильного поведения инструментов и компонентов, измеримо.

Таким образом, основной проверяемый тезис – можно ли увеличить непрерывность и продуктивность процесса разработки за счёт только технических усовершенствований, и на какую величину.

Дополнительно рассмотрен новый, экспериментальный инструмент для интерактивного программирования. Инструмент позволяет вносить глубокие изменения в программу в процессе её работы, без перезапуска. Исследуемый вопрос: минимизация длительности цикла программирование-компиляция-тестирование-коррекция и проверки небольших частей кода, в частности, усовершенствование рабочего процесса отладки за счёт совместного использования интерактивного редактора и отладчика.

Ключевые слова: кросс-платформенный, программирование, удобство пользования, оптимизация.