UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
ELABORATO

MODERN SOFTWARE METRICS: DESIGN AND IMPLEMENTATION

RELATORE: Ch.mo Prof. Carlo Ferrari

LAUREANDO: *Enrico Ros*

Padova, 26 Marzo 2010

Contents

Abstract

As software becomes more complex, new measurements methods are needed to leverage quality, improve user's experience and reduce energy consumption.

This dissertation introduces the *Inspector*, a software tool for framework-level dynamic software analysis and the *Thermal Painting*, a new software metric for measuring the performance of the graphical subsystem of a program.

The *Inspector* breaks many of the constraints that affected traditional tools like debuggers and function-level profilers, like the need to alter the source or binary code and the impossibility to profile already running code that exhibits bad behavior, and provides a unique work environment for conducting the tests.

The *Thermal Painting* is the new software metric that measures the per-pixel energy required to paint a graphical user interface, allowing to profile and improve the graphical performance of a program.

Chapter 1

Dynamic program analysis and motivations for new tools

Contents

Everybody makes mistakes, in life, at work, in every branch of art and technology. The more complex the subject gets, the more the probability of incurring into some unwanted condition raises.

Software engineers are often challenged with a really big task that is doing a perfect product, fully featured but fast, correct but quick to write, working good for that particular job but still adaptable to similar jobs. Actually it is a hard task because in software just a single misplaced bit could break everything and make things crash in chain. Unfortunately software does not work like that machinery that can lose bolts and still work.

Being immaterial, software can be virtually duplicated in millions of copies in a very short time and deployed to a vast user base in a matter of minutes, over private networks or over the Internet, thus increasing the magnitude of the damage a single error can do.

The times in which we live are challenging for any product maker because people have high expectations. It is not enough to buy a product that "just works", it must work good, be appealing to the eye, weight less, last longer. Business is trying to respond to this needs with management strategies that allow for less than 5 defects per million, such as the Six Sigma method [1].

To comply with those strict requirements, even the software development methodologies needed to be adapted. From signed reviews by seniors or peers to pair programming, to agile development methods, to smart content versioning systems, to regressions test suites, to automatic building farms, the focus is shifting from man-driven to machine-driven. The more you take repetitive tasks off the programmer and assign them to the machine the more you can reduce errors.
The very strict nature of some programming languages such as `C++`, which will be the main focus of this dissertation, while on one side constraining the flexibility available to the software engineer, on the other side it allows for automatic tools that look for mistakes. Computer science has been involved on this testing tools since its beginnings, and lately the topic has gained even more focus for the reasons outlined above.

In recent years more sophisticated tools, that help to increase the quality, have seen the light, like:

continuous integration tools: Those tools allow for automatic merging the changes, building, packaging, testing and deploying the software written within teams. Their main task is testing each and every source code change to see whether it breaks the build or introduces regressions or security issues.

Usually they are quite simple scripts running over very powerful machines.

regression tests suites: Those are collections of tests, in form of scripts or executables, each one testing a very specific feature of the target product. Usually written by the same people developing the main product, they can be used either for validating a feature (as in *test-driven development*) or for checking that a behavior will not change over time (as in *regression testing*).

static code analyzers: The analysis performed by those tools is done over the source code of the program, without building and executing it. By just looking at the source code those analyzers can find a wide range of issues, from potential security problems to proving mathematical properties of the code. The way they achieve this, varies from simple pattern matching over each line of code to more complex "just in time" compilation and execution of code paths.

dynamic program analyzers: Those tools analyze the software by executing programs built from that source code. The programs can be executed on real or virtual processors. Usually those tools look for generic execution issues, like out of bounds memory accesses or memory leaks. Sometimes they require the source code to be recompiled by adding some compile-time instrumentation.

profilers: This is a subclass of the *dynamic program analyzers.* Those tools collect information from the program as it executes. The gathered information can then be used to optimize the program, whether this means optimizing it for *speed*, *size*, *startup time*, *screen area*, *memory usage* or any combination of the above.

The work presented here is improving the *dynamic program analyzers* class by adding completely new and unmatched levels of functionality.

1.1 Prior art

Here we identify three kind of programs available on the market that allow, to some degree, to do performance analysis and improve the software. Those are debuggers, profilers, and specialized profilers.

1.1.1 Debuggers

In the beginning there was the *debugger*, a computer program designed to test other programs. Debuggers are used for finding and reducing the number of *bugs* (or "unwanted behaviors"), using a well defined methodology tightly related to the programming language the source code it is written into. However there are some features that are quite common among debuggers:

stopping execution: When some user defined or self-raised condition is met, the debugger stops the program showing the position (in terms of source code line or machine code line) the program has reached.

showing back traces: This is a list of machine code ("stack addresses") or function names that led the program to the position it is. The list usually starts at the "main" function and can be really long and sometimes complex to understand, for example in case of event loop dispatchers or recursive functions.

stepping: This is the process of advancing the execution by one line of either source or machine code. This is usually needed when looking for the exact point in which some condition happens. A few modern debuggers allow reverse stepping that is "going back in time", or reverting the effects produced by the execution of that instruction.

displaying memory: This is the ability of showing the contents of the memory allocated, or used, by the program. Depending on the programming language the debugger can relate the memory addresses to function parameters, global or local variables and exceptions. Usually the ability to alter values is offered.

The debugger is the first testing tool available to the developer and the one fulfilling the very basic needs, as outlined above. It is the tool to use when the program crashes or when stepping through each line of code is the only way to find the answer you are looking for.

Debuggers are a solid and stable technology and there is a wide range of similar products available in the market, from open source solutions to commercial ones.

1.1.2 Profilers

It is not enough for a program to be correct. When the "quality" bar raises, debuggers lose effectiveness. When looking for a way to improve a specific quality of a program like *speed* or *memory consumption*, a new program class comes into use: Profilers.

Profilers collect data from an executing program. The execution can happen over a real processor or over a virtual one, in which the execution of every instruction is emulated in software. For gathering data the profiler needs an instrumented program. From the one requiring more manual intervention to the more automatic one, here is the list of the ways to instrument a program for profiling:

- manually performed by the programmer over the source code, for example by calling a function in strategic places.
- performed by automatic tools over the source code (basically an automated version of the above).
- generated by the source code compiler when emitting the object code.
- added to the object code by binary translators.
- performed during execution by virtual executors.
- injected during execution by specialized programs.

This is also the order in which the instrumentation methods were introduced over time.

Typically the data gathered during the profiling consists of: duration of function calls, frequency of function calls, memory usage, cache failures. The data is then analyzed by the programmer to better direct the optimization efforts. By looking at the numbers, knowledge can be gained about: functions that take too much time to execute, functions that are called too often, where and how much memory is allocated, where "cache misses" happen and how to avoid them.

The main disadvantage of profilers is that they lack the knowledge of the program, of what it is supposed to do, how it works, how the underlying framework works, how this is translated into machine language. The best they can do is to make assumptions about the underlying ABI[2], so the best they can tell is which function is called and for how long.

In the next chapters we will see how developing a profiler that has more knowledge about the code it is executing will lead to much better results.

1.1.3 Specialized profilers

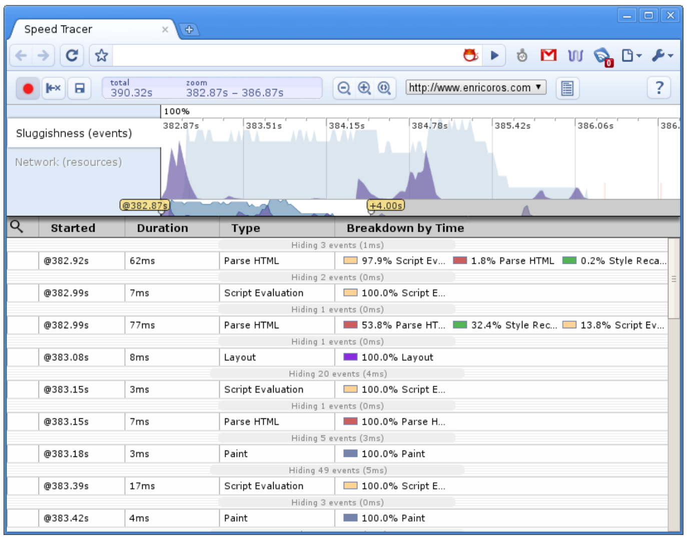
Profilers with high knowledge of the environment they operate in, fall in this category. The next examples highlight very recent work on this field. One is a profiler embedded in the containing application, the other one aims to be a "framework profiler".

Google Chrome's Speed Tracer

Speed Tracer is a tool that helps identifying and fix performance problems in web pages and web applications in general. It displays metrics that are taken from low level instrumentation hooks inside Google Chrome[3].

Speed Tracer (figure 1.1) can be installed and used as an extension of the Google Chrome software. This popular web rendering engine has been instrumented on purpose, to allow profiling of the web pages loaded, layouted and rendered by the web browser itself. The instrumentation can be enabled when starting the browser, since enabling it by default would mean to lose little performance even when not profiling.

Speed Tracer was built for two main purposes: to allow Chrome's developers to profile the web rendering engine itself, and to allow the whole world wide web community to build better web pages. The theory behind this is that by just giving web developers a cue about where performance is spent when executing their web applications, this will help them identifying any wastes, optimize resource loading, and build cleaner and faster web pages. To do this, Speed Tracer gives



Started	Duration	Type	Breakdown by Time
		Hiding 3 events (1ms)	
@382.92s	62ms	Parse HTML	🟧 97.9% Script Ev... 🟥 1.8% Parse HTML 🟩 0.2% Style Reca...
		Hiding 2 events (0ms)	
@382.99s	7ms	Script Evaluation	🟧 100.0% Script E...
		Hiding 1 events (0ms)	
@382.99s	77ms	Parse HTML	🟥 53.8% Parse HT... 🟩 32.4% Style Rec... 🟧 13.8% Script Ev...
		Hiding 1 events (0ms)	
@383.08s	8ms	Layout	🟪 100.0% Layout
		Hiding 20 events (4ms)	
@383.15s	3ms	Script Evaluation	🟧 100.0% Script E...
		Hiding 1 events (0ms)	
@383.15s	7ms	Parse HTML	🟥 100.0% Parse H...
		Hiding 5 events (3ms)	
@383.18s	3ms	Paint	🟦 100.0% Paint
		Hiding 49 events (5ms)	
@383.39s	17ms	Script Evaluation	🟧 100.0% Script E...
		Hiding 3 events (0ms)	
@383.42s	4ms	Paint	🟦 100.0% Paint

Figure 1.1: The Speed Tracer plugin of Google Chrome showing the performance of a website.

information about time spent in:

- Javascript parsing and execution
- Layout
- CSS style recalculation and selector matching
- DOM Event handling
- Network resource loading
- Timer fires
- XMLHttpRequest callbacks
- Painting

Since its introduction, Google Chrome has been gaining momentum and as of March 2010 it is the browser that is having the biggest gains with an estimated market share of 5% to 16%. Chrome is using the same Webkit [4] open source web engine that Apple, Nokia, Palm and RIM are using, making Webkit the most used browser in the mobile market.

This is an example of how profiling and making a faster browser and a faster web is attractive to both vendors and users.

NVIDIA's Parallel Nsight

Parallel Nsight (codenamed "Nexus") [5] is a GPU profiler and debugger built by NVIDIA for Microsoft Visual Studio 2008. It is the first complete solution for developing programs on the GPU, providing tools that allow:

- System performance analysis
- Debugging CUDA C
- Debugging graphics (OpenGL and Direct3D)
- Debugging processing (OpenCL, DirectCompute)

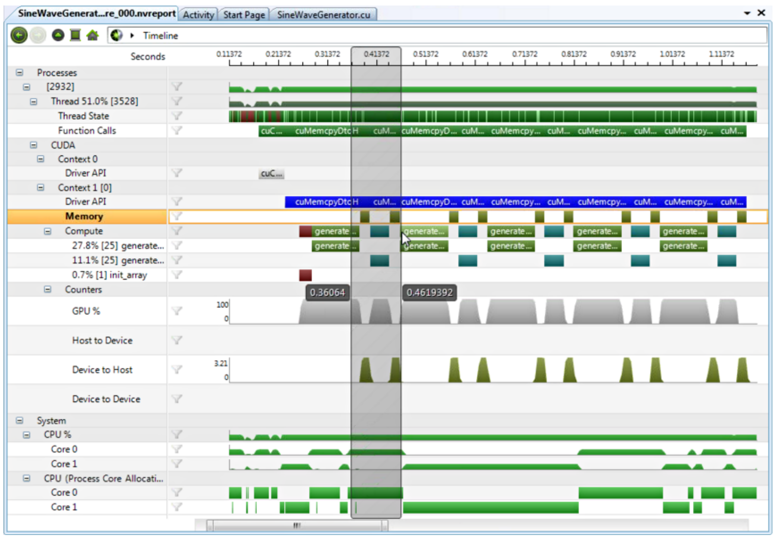


Figure 1.2: NVIDIA Parallel Nsight showing profiling counters.

The profiler (figure 1.2) collects data from the GPU internal counters and plots it over a time-line where it is easy to see operations with the nanosecond resolution and to find out the exact execution and data transfers order.

Parallel Nsight is a recent example (announced Sept 30 2009) of a specialized profiler that operates at the "framework" level. It works by using some facilities provided by the NVIDIA operating system graphics drivers and it runs where an NVIDIA graphics board is available. Even if it dictates a strong vendor lock-in, the features it provides are completely new an much needed by a development community used to treat graphics boards as black boxes which execute code and are impossible to debug. In this regard, Parallel Nsight opens up many possibilities. We will see how many more can be opened by operating at the same level.

1.2 Modern challenges

In the past decades, the "golden age of silicon" writing a program that performed good was not a priority, since everybody knew that every 1.5 years the performance of integrated circuits would have doubled. This drove the need for more processing power, more graphical power, more bandwidth, more storage space, higher screen resolution.

This exponential growth could not last forever and in recent years there have been a number of regressions on Moore's law. Intel's Atom, a CPU designed for netbooks, is sacrificing performance to power consumption. People are asking for cellphones that have more functions ("smartphones") but must work longer.

1.2.1 Power efficiency

A change of direction in development has happened: the software must work fast on cheap slow hardware and it must consume the lowest possible energy to do what it is supposed to do. If the software wastes too much resources it is not going to succeed in the mobile market.

This need is driving some shifts:

programming languages: To write power efficient code, the developer must work as close as possible to the machine, thus he moves to the classic `C` or `C++` programming languages and to more efficient frameworks (and software stacks).

programming practices: The developer requests and allocates resources manually and releases them as soon as they're not needed.

program optimizations: Code is profiled with standard call profilers to spot where to operate to increase throughput and remove bottlenecks.

There are currently no good tools available for the average developer aiming for efficiency. There are either low level machine debuggers, generating gigabytes of hard-to-parse data per second or generic profilers counting function calls. Providing a tool to measure the performance, or power consumption, in an *intuitive* way would mean enabling programmers to really spot where the performance is lost and selectively perform optimizations there.

The question that needs an answer is: **what can be done to reduce the power consumption of a given application?** In chapter 3 we will give some answers to this question.

1.2.2 Code complexity

As the code complexity raises there is a strong need for new new tools.

Nowadays even the cellphones run operating systems and frameworks built out of millions of lines of source code. All the popular open and closed source desktop environments are at least that huge and they are made by a myriad of components that interact closely together. As the size increases, more code is reused from past projects, bought from third parties or borrowed from the open-source world. This means that a software system running on a cellphone is written by thousands of different people, in different teams, in different countries and with different programming skills and priorities.

In this complex and often fragile environments it's easy to see that debuggers and standard profilers won't provide the answers the developer is looking for. Halting one piece of code on a breakpoint might crash the other pieces, stepping line by line could lead to nowhere, time spent on functions will not tell if that code path could be done better.

What is needed here is a way to transform the software in *intuitive* ways to allow new kind of measurements and comparisons. The transformations should be based on easy to grasp concepts such as *weight*, *size*, *temperature*, *space*, *time*.

The question that needs an answer is: **how can we measure software in new intuitive ways?** In chapter 3 we will give some answers to this question.

1.2.3 Old questions still unanswered

There are specific questions that every developer has asked himself at least once. If this happened is because some questions survived time, and that means that nobody provided the answers.

Those questions are:

What is my program doing? The short answer is: exactly what it is programmed to do. The CPU is executing the binary code loaded by the operating system bit by bit. Entire stacks of software like kernel code or toolkit libraries are involved when running a simple program made by a developer. So when this question is raised the user feels to have lost the grasp over the running program, and the higher the abstraction degree is, the more the developer will feel the need for an answer.

Why is it so slow? There is no such thing as “slowness”. There are finite resources, like the processing power of the system, and the program has to make the best use of them otherwise it will behave in suboptimal ways. A perfect answer to this question would be to identify the bottlenecks of the application and how they adversely affect the program execution speed.

Why did it become so... ? This questions deals with *the change.* Like physical objects, over time some programs can deteriorate and a number of different issues that were too small to be noticed when the program was just started can become more evident after hours, days or years of operation. Having some ways to check the program at intervals and highlight the changes could be helpful in resolving this kind of problems.

While in general sense it is not possible answer all those questions, it is possible to do it under certain circumstances. In chapter 3 we will give some answers to those questions.

1.3 Inspector and the modern software metrics

The work outlined in this dissertation is the response to a real need, the need for some answers to the questions above. Due to the lack of available tools, we had to create something new, a new software tool to allow specialized profiling at a new level, the Framework one.

This is why *Inspector* was born, but it is just the platform, the technology that enables new kind of tests, or *metrics* as we will call them on chapter 3. While Inspector provides the way to access the program, with all the knowledge to access the software stacks behind it, the *modern software metrics* that were developed over it allow for new kind of measurements. They give the real answers to the questions in section 1.2.

In the next chapters we will take a look at Inspector, answering *why*, *what* and *how* it was created. Then on chapter 3 the design process for creating the modern software metrics and the *Thermal Painting* metric will be explained, highlighting the significance of the results that can be obtained by applying those new methods to software development.

Chapter 2

Inspector

Contents

INSPECTOR [6] is a tool for *framework-level specialized profiling.* This program is the answer to the modern challenges listed in section 1.2 and breaks many of the constraints that previous tools had. In this chapter we will analyze the design that came out of the requirements and see how it was implemented.

2.1 Features

Inspector has been designed and implemented to **address some needs that currently available tools or profiling frameworks do not provide**. In particular those needs are addressed:

profiling of unmodified source code: If the code must be changed before profiling then you are not profiling the original code, but something else. Plus you want to be able to take an existing program and profile that even if you have no access to the source code.

profiling with unmodified system libraries: Similarly to the point above, we want to be able to profile the program in its environment, without requiring changes to the libraries or frameworks it is using.

profile running executables: It may happen that a program starts acting in unexpected ways after many hours of operation. In this case you do not want to close it and start it again in a new profiling session. You want to be able to profile *that exact running instance.*

different back-ends: A running code may use any number of system *APIs* (or *frameworks*, *toolkits*, *libraries*) and Inspector must be able to support specialized profiling for a number of those.

modern user interface: The interface must be able to support many Inspection sessions, must be intuitive and easy to operate, must show the status of all the running operations, must be able to load, save and compare data sets.

Inspector does all of the above and more, but before going deeply into the design details, let us introduce some terms that will be used throughout this chapter.

2.2 Definitions

Here are some definitions that will be used throughout the rest of the chapter.

Inspector: This tool.

Target: Any running executable.

Probe: Machine code that can be injected into the *Target*.

Runtime Injection: The operation of inserting the *Probe* into the *Target*.

Framework: An API, framework name, toolkit or system library.

Test: A specific test, probing operation or measure for a *Framework*.

Backend: An *Inspector* component, dealing with a specific *Framework*.

Backend Module: A *Backend* component implementing one or more *Tests*.

Backend Panel: A graphical user interface to control a *Backend Module*.

Inspection: A probing session with a *Backend* connected to a *Target*.

Inspector Dashboard: The *Inspector* control panel allowing to operate on *Inspections*.

In the definitions above, some architectural elements are introduced. The order in which the items are presented accounts for the relations of the terms too.

So with no further hesitation let us look at how things work.

2.3 Principles of operation

First let us understand the **key principle** behind Inspector. It is able to do all it does because of the way the operating systems are made. Operating systems provide entire stacks of software from low level (e.g. I/O functions) to high level (e.g. web tookits) and the way applications use them is via **symbolic resolution**.

When using a Framework, the related machine code is contained in binary files prefixed with indexes that tell where to find the “public” or “exported” symbols. This tiny detail allows Inspector to do Runtime Injection of the Probes into the running executables (Targets). The Probes on their part are binary code that is loaded when the Backend needs to handle the Target and uses symbols of a specific Framework to operate. A probe can just *invoke* specific symbols or operate in more complex ways like integrating into the Target's object code.

Inspector is then a generic tool using Backends that know exactly the way the Framework they are profiling works. And those Backends are able to watch or control in some ways the operation of the Target.

The way the Backends and Probes work is strictly dependent to the Framework. A common way they work is presented in chapter 2.5.3 where we will analyze in detail a Backend implementation.

2.4 Inspector design

Inspector has been designed to meet all the requirements described in section 2.1.

From a bird's-eye point of view, Inspector lets the user pick a Target, select which Backend to use and then starts analyzing that pair. The analysis is then completely Backend and Target dependent, so Inspector only provides some facilities and mandates design patterns to the Backend developers.

This section focuses on the design architecture of Inspector while section 2.5 will focus on the implementation details.

2.4.1 Software architecture

The relevant parts of the code architecture of Inspector are shown in figure 2.1.

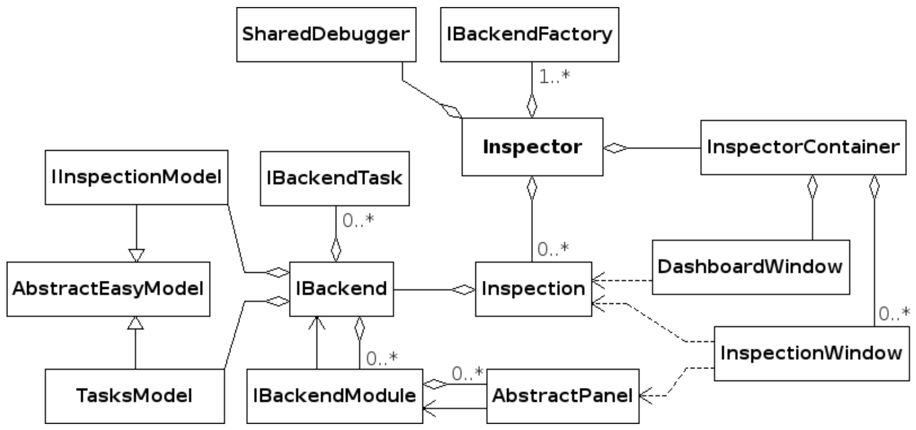


Figure 2.1: The software architecture of Inspector. The core functionality (IBackend and related classes) is greatly separated from presentation (DashboardWindow and surrounding classes).

The entry point for figure 2.1 is the *Inspector* class. It owns:

- a *SharedDebugger*, an utility class that encapsulates the functionality of a debugger and may be used by Backends. See section 2.5.3 for more information about this class.
- multiple *IBackendFactory*, that describe the features of the Backends and can create one on demand. This class will be described in section 2.4.2.

- an *InspectorContainer*, the user interface entry point, described below.
- can have multiple *Inspection*, one for each profiling session.

The *InspectorContainer* holds the complete user interface subsystem. It owns:

- a *DashboardWindow*, that allows to start new Inspections and shows information about all the running ones. This class and the following do not access the Backends directly, they just have access to data models or graphical user interface components.
- can have multiple *InspectionWindow*, each one interacting with a different inspection. This class and the previous one are described in section as they are defined by their implementation.

The *Inspection* class describes a running inspection and exists just before the start of the Inspection and right after its end. It's used mainly by graphical user interface components. It owns:

- an *IBackend* instance, with a Backend set up and probably connected to the Target. For more information see section 2.4.2.
- holds strong references to the *IInspectionModel* and *TasksModel* classes that are created by the Backend.

The *IBackend* class operates over the Target, can do Runtime Injection, assume that the Target makes use of the Framework it is made for, use debugger functions and all the results of its operations are stored in data models. It owns:

- an *IInspectionModel*, a data model that holds the common data about the current Inspection and is usually referenced from the graphical user interface components.
- a *TasksModel*, a data model that holds brief information about the running tasks and is used in the same way the above model is used.
- can have multiple *IBackendModule*, each one providing the real testing/probing functionality over a Backend. See section 2.4.3 for more information about Backend Modules.

- can have multiple *IBackendTask*, each one describing a task that is running on the Target. See section 2.4.5 for a detailed description about this.

The naming of the classes has been chosen to reflect their nature. Classes starting with the capital “I” letter are meant to be interfaces and they are meant to be reimplemented and specialized. Those classes are *IBackend*, *IBackendFactory*, *IInspectionModel*, *IBackendModule*, *IBackendTask*. The exception to this rule is the *AbstractPanel* class that violates the pattern because it belongs to a different abstraction than the one containing the classes mentioned above (see section 2.4.4).

Inspector classes, their relations and their visibilities define some patterns of operation that are not visible in the architecture diagram. Those patterns are enforced for ensuring good programming practices, to meet the requirements of the design, and to avoid violations over the existing structure.

In the next sections we will see those patterns explained.

2.4.2 Backends

The Backend abstraction provides Inspector the tools it needs to analyze the behavior of a Target that uses a specific Framework. Each software stack used by a running program offers usage semantics and entry points that allow for a Backend to be made.

Backends can be made, for:

libraries and frameworks: such libraries can be used for doing measurements over the code using them. Examples of this include the Qt Backend explained in section 2.5.3.

hardware instrumentation: this can be used to measure Targets running on that hardware. An example of this is the NVIDIA$^{\text{TM}}$CUDA$^{\text{TM}}$Backend using the hardware performance counters that are enabled when profiling CUDA applications.

bytecode interpreters: bytecode written for specific interpreters allows for low level profiling Backends to be made. Those will use features specific to the interpreter to offer information about the profiled target. An example of this could be a Backend for the Microsoft Common Language Runtime, for Java bytecode execution or for Adobe Flash programs.

virtual machines: there are virtual machines providing information about the Target they execute and this information could be parsed by a Backend. An example of this could be a Backend for the Valgrind[7] virtual machine, a software tool providing memory debugging, memory leak detection and profiling functionalities.

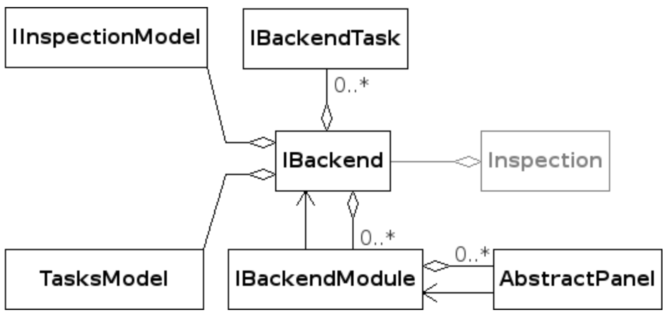


Figure 2.2: The IBackend interface and the related classes.

Backends in Inspector are defined by the *IBackend* interface and the related classes as show in figure 2.2.

The *IBackend* class provides virtual methods to start the Inspection and create *AbstractPanel*s (see section 2.4.4) and it owns data models, Backend Modules and task descriptors.

The group of classes shown in figure 2.2 is made in a way that allows *layered abstraction*. Since the *IBackend* class must give outside access to some of the internal classes, like data models or task descriptions, those classes can be extended too. This way the *IInspectionModel* class can contain Backend-specific data, that is used by the Backend, the Backend Modules and the tasks while providing generic data to the outside watchers. A Backend implementation must at least re-implement the *IBackend* class providing some needed functionality and then it can re-implement the data models, modules, tasks. While the Backend group of classes has the complete visibility over all the Backend extensions, the outside classes will still be insulated by the specializations made in the group and only use the operations and properties of the base classes.

The Backend Modules, described in section 2.4.3 allow to extend the Backend with new functionality. The Backend creates and registers all the Backend Modules at runtime.

The *IBackendFactory* (shown in figure 2.1) is used to describe and instance *IBackend* classes. That class uses the Factory pattern [8] to allow for the creation of objects that may require complex setups and are not known to the Inspector base architecture. This way Backends can be *plug-in*s too.

See figure 2.6 for a sample Backend implementation.

2.4.3 Backend Modules

Backend Modules are used to provide the Tests to the Backends. They exist to allow cleaner code separation, not for a real need of "pluggable" code even if they could be made plug-ins.

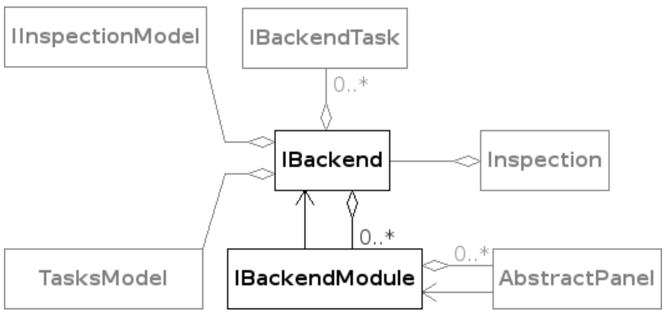


Figure 2.3: The IBackendModule interface and the related classes.

The Backend Modules are made for a specific Backend. They use the functionality provided by that Backend, usually communication or Runtime Injection facilities, to do higher level Tests. They don't have to deal with low-level details, because this is the job of the Backend, but they must implement the logic that stands behind a certain Test, the related data models and the views to display and interact with that data.

The Backend Modules do:

- access the Framework dependent functionality provided by the Backend.
- describe the Test, or Tests, they implement.
- create *AbstractPanel*s to start new Tests and display the results.
- hold any data model specific to the Tests they implement.
- can create specialized *IBackendTask* to do their job.

Often a Backend Module provides the Backend Tasks needed to perform the job. So usually the Backend Tasks are assumed to belong to the Backend Modules. More information about the Backend Tasks is in section 2.4.5.

See figure 2.6 for a sample Backend implementation.

2.4.4 Backend Panels

The Backend Panels are used for providing the user a way to interact with the Tests. Those Panels act like Views and Controllers in the MVC pattern.

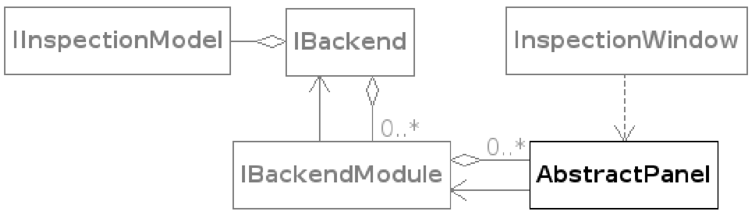


Figure 2.4: The AbstractPanel interface and the related classes.

*AbstractPanel*s are created by Backend Modules when a Test is selected from the tests menu of Inspector and they are plugged into the *InspectionWindow*. Upon creation they are wired to the parent Backend Module and to the data models they will use (may be *IInspectionModel*, or models of the Backend, or models of the Backend Module).

When re-implementing the *AbstractPanel* class, the specialization needs to add GUI components, like indicators, knobs, buttons, tables or more complex views and connect those components to the functions of the Backend Module, and to update the graphical components whenever the data of the models changes.

Since only one *AbstractPanel* can exist per Inspection, and the panels may change while there are Test running, it is important to behave good in terms of consistency. So when a panel is created it must be updated to reflect the current state of the model allowing to switch back and forth between different panels without seeing changes when switching back to the same panel.

2.4.5 Backend Tasks

Backend Tasks have been introduced to provide a good user interaction with Inspector. They allow for Tests starting, queuing, cancellation and parallel running while reporting information to the user and avoiding to lock the user interface.

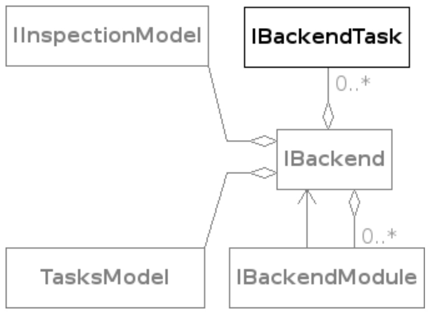


Figure 2.5: The IBackendTask interface related to the surrounding classes.

The *IBackendTask* does:

- perform the Test.
- inform the user about its activation and progress.
- behave asynchronously.
- feed data models.

It works by accessing the Framework dependent functionality provided by the Backend, that is available to the Task for its whole activation time.

There is an internal state machine that drives the activation and deactivation of the tasks and forces a Backend Task to play nice. A Backend Task can request activation, and it can be granted, refused or delayed. At any time it can receive the request to interrupt/deactivate the task. When the Backend Task ends, either because it wasn't allowed to start, or it has been canceled, or more likely it completed its job, it flushes out its data and notifies the Backend it finished.

Figure 2.5 shows how the Tasks interact with their neighborhoods.

2.5 Implementation in Qt Creator

Inspector [6] has been implemented in Qt Creator [9] by Nokia.

At the time of writing the Inspector plugin for Qt Creator is at version 1.3.82, it implements the Inspector architecture, adds a Qt Backend and some Qt Backend Modules that implement some Tests (detailed in chapter 3).

The plugin is made of nearly 11000 lines of code, it has been written in C++ using the Nokia Qt framework [10].

This section focuses on all the implementation details of the design detailed in section 2.4 and on all the practical things needed to create Inspector.

2.5.1 What is Qt Creator

Qt Creator is a cross-platform C++ integrated development environment which is made by Qt Development Frameworks a subsidiary of Nokia.

Inspector has been implemented as a Qt Creator plug-in because this IDE provides many useful features, such as:

- it is written in C++ with the clean Qt libraries.
- runs on Linux, Apple Mac OSX, Microsoft Windows.
- it is open source.
- provides a plug-in framework allowing to add features and modify existing behaviors easily.
- wraps debugging functionalities in a convenient way.
- already provides ways to load binary code at runtime.

This excellent IDE is available free of charge and under the open source license agreements so everyone is able to contribute. It is used widely and has a very vibrant community around: during the eight-months development cycle of the

Inspector plug-in, many other plug-ins were added in the official code base. Qt Creator is developed in the open in a *git* software repository on:

`http://qt.gitorious.org/qt-creator`

2.5.2 Implementing the Inspector plug-in

The Inspector plug-in for Qt Creator realizes the architecture described in section 2.4. The realization is straightforward, mapping one class for each of the design elements outlined above. In addition to that there are some utility classes used for better integration with Qt Creator, some shared utility classes and some workarounds for constraints imposed by the present structure of the IDE.

The most relevant implementation constraints that had to be defeated are:

resource locking: given the fact that there are finite amount of resources (e.g. the debugger that exists in single instance) those must be wrapped and shielded by a resource borrowing/locking mechanism.

asynchronous communication: since the communication with the Target is asynchronous no assumptions can be made about its state. This led to the choice of using asynchronous states machines where appropriate.

gap-less graphical user interface: the user interface must not lock, the operations could be canceled, and the user should be aware of what is happening under the hood.

playing nice with other plug-ins: the Inspector plug-in must behave well and avoid disruptive changes to other plug-ins or to the shared classes and data. The good citizenship principle poses some constraints to the operation.

The file-system structure

The Inspector plug-in is located in the `src/plugins/inspector` folder inside the Qt Creator source code.

Folder	**Contents**
inspector	folder for the Inspector plug-in
inspector/{files}	Inspector plug-in files
inspector/backend	folder for a Backend
inspector/backend/{files}	Backend implementation files
inspector/backend/test	folder for a Backend Module
inspector/backend/test/{files}	Backend Module implementation files

Table 2.1: Contents of the Inspector plug-in folders.

As shown in table 2.1, Inspector implementation files lie in the base plug-in directory. On each sub-directory lies a Backend implementation along as all the needed Backend files. Each sub-directory of a Backend folder contains all the files defining a Backend Module. If some Backend or Backend Modules share some functionality, that functionality should be placed on the previous level.

The build system follows the same scheme, using a single project file in the Inspector plug-in folder that includes one project file for each Backend sub-folder. The Backend file includes all the file to build that Backend and the children Backend Modules.

Base file name	**Class name**	**Description**
abstracteasymodel	AbstractEasyModel	base class for data models
abstractpanel	AbstractPanel	interface for Backend Panels
dashboardwindow	DashboardWindow	gui: the Dashboard
ibackend	IBackend	interface for Backends
ibackendmodule	IBackendModule	interface for Backend Modules
ibackendtask	IBackendTask	interface for Backend Tasks
iinspectionmodel	IInspectionModel	extendable Inspection data model
inspection	Inspection	describes an Inspection
inspectiontarget	InspectionTarget	describes a Target
inspectionwindow	InspectionWindow	gui: the Inspection window
inspectorcontainer	InspectorContainer	gui: the main window
inspectorplugin	InspectorPlugin	the entry point
inspectorrunner	InspectorRunner	integration: run targets

Base file name	**Class name**	**Description**
inspectorstyle	InspectorStyle	integration: appearance
modulemenuwidget	ModuleMenuWidget	gui: the side menu widget
panelcontainerwidget	PanelContainerWidget	gui: a container
plotgrid	PlotGrid	gui: a charting function
probeinjectingdebugger	ProbeInjectingDebugger	integration: debugging
runcontrolwatcher	RunControlWatcher	integration: current runs
shareddebugger	SharedDebugger	integration: debugging
singletabwidget	SingleTabWidget	gui: top bar
statusbarwidget	StatusBarWidget	gui: bottom bar
tasksmodel	TasksModel	the Tasks data model
tasksscroller	TasksScroller	gui: display tasks status

Table 2.2: Inspector implementation files and classes.

In table 2.3 there is the list of classes that define the Inspector plug-in.

The relations between the classes have been described in section 2.4, for a better insight see the publicly available Inspector source code [6].

2.5.3 Implementing the Qt Backend

Inspector is a good tool but it is useless without Backends (see section 2.4.2).

The first Backend that was made is the one for the `Qt` Framework. Following the good software principle of "eating your own dogfood", providing a Qt Backend allowed to *use Inspector to profile Inspector*.

Profiling the Qt Framework

The first step for creating the Backend was to analyze the Qt libraries to find out candidate entry points, possible hooks and static public symbols that Inspector could take advantage of. Having access to the open Qt sources was really helpful in this preliminary phase. While there is *no public instrumentation support in Qt*, the way the Framework is made allows:

hooks on events dispatching: there is a private callbacks mechanism, defined in the `QInternal` class (*qobject.cpp*) that allows to set callback functions when running the event loop. Since the event loop is the engine that dispatches all the user, network, windowing system, and timing events in a modern application, having callbacks over that allows to keep the control of the Target.

hooks on signals/slots: using the `QSignalSpyCallbackSet` structure (qobject_p.h) and the `qt_register_signal_spy_callbacks` function allow for callbacks on signal emission and slot activation. Signals/Slots is a language construct introduced in Qt, which makes it easy to implement the Observer pattern. This concept has been adopted by other toolkits such as *boost* signals and *C#* events and delegates.

introspection: classes that inherit from `QObject` are introspectable. This allows any external watcher to retrieve objects hierarchies, operate on properties and list and invoke methods on the object. This feature alone allows many kinds of manipulations on a Target that uses the Qt framework. The software metrics outlined in 3 are using the introspection.

use of all the Qt event-based classes: any code Runtime Injected by the Qt Backend can create instances of all the classes the Target has access to. That means being able to create or extend Gui/Network/Core components and to add them to the event loop to take part in event dispatching.

The Qt Backend makes some use of those functionalities on the Target. To be able to do so, it needs the following:

1. to *load object code* into the Target
2. a *bi-directional communication channel* with the Target

Code Injecting Debugger

The first need is addressed by Qt Creator itself. The Debugger plug-in allows to use the *DebuggerManager* class that provides debugging facilities on all the supported operating systems. In particular the GNU GDB debugger and the Microsoft CDB debuggers are supported. There are two minor problems with the Debugger plug-in:

1. the user can interfere with the debugger itself (e.g. stopping it by clicking the stop button while Inspector is using it).

2. there only is one debugger and it may be already in use (by the Inspector itself or may be busy with some debugging session).

The *ProbeInjectingDebugger* class solves the first problem by providing the functionality needed by Inspector while monitoring other possible accesses to the debugger by third parties. This class allows to load object code (in form of shared libraries) when starting a new Target or attaching to an existing one, plus it allows to invoke loaded functions by their name.

The *SharedDebugger* class solves the second problem by hiding the debugger and offering it as a “resource”, so it can be acquired and released by only one user at the same time.

Communication with the Qt Target

In the Qt Backend the “communication” from the Backend to the Target is done by calling functions on the Target. They can be functions provided by the Qt Framework, or functions provided by the Target (really unlikely, since the Backend only knows about the Framework), or provided by the injected Probe.

The communication from the Target to the Backend happens through a local socket. Both parties are involved to establish the communication, that happens in this way:

- the Backend starts a server that listens for incoming connections.
- the Probe is injected in the Target.
- the Probe gets the server name (via a function call on the Probe).
- the Probe connects to the server and keeps a *synchronous* streaming channel opened.

This sequence of actions introduces some details about the Probe, which will be described in detail in section 2.5.4.

Qt Backend structure

The Qt Backend is made of two components: the Probe and the Backend itself. The Backend follows the guidelines outlined in section 2.4.2.

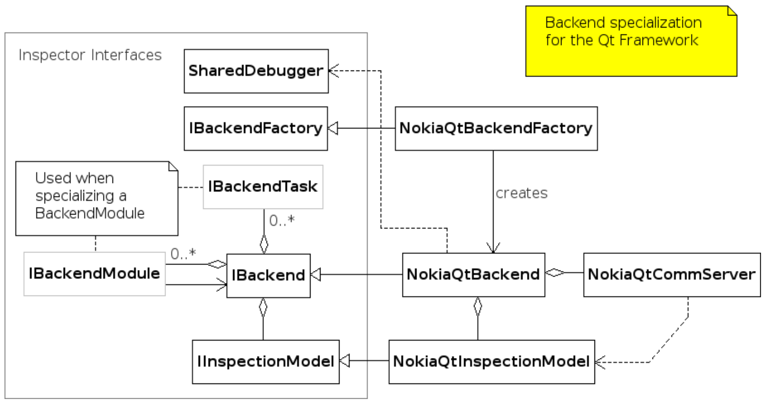


Figure 2.6: The Qt Inspector Backend architecture.

In figure 2.6 there are the main classes that make the Backend. The *NokiaQtBackendFactory* creates a *NokiaQtBackend* instance when a new Inspection is started over a Qt Target. The Qt Backend then creates the *NokiaQtCommserver* that will handle the incoming communication from the Probe. The *NokiaQtInspectionModel* class extends InspectionModel by adding some data that can be used by the Qt Backend Modules. In particular, the extended data is used by the *Info* Backend Module to show the status of the Inspection.

Base file name	**Class name**	**Description**
datautils	DataUtils	data manipulation
nokiaqtcommserver	NokiaQtCommserver	probe communication
nokiaqtbackend	NokiaQtBackend	Qt Backend
nokiaqtinspectionmodel	NokiaQtInspectionModel	extends the data model

Table 2.3: Inspector implementation files and classes.

In the Qt Backend folder there are a number of sub-folders, one for each Qt Backend Module. At the time of writing, the following modules are present: *Anomaly*, *Blueprint*, *Crasher*, *Events*, *Heartbeat*, *Info*, *Object*, *Painting*.

2.5.4 Implementing the Qt Probe

The Probe is the critical part of the Nokia Qt Backend. It provides the communication channel with the Target and contains all the object code needed to hook into the framework (as described in section 2.5.3) and to perform the measurements defined in chapter 3.

The Probe is a small shared library written in `C++`. It has few public functions and is forcing the "`C`" symbols mangling to allow the *ProbeInjectingDebugger* to call functions.

When writing the Probe it is necessary to follow some rules to avoid breaking the Target:

- the symbols must be carefully named to avoid polluting the namespaces and to avoid clashing symbol names.
- the probe operation should have minimum impact on the target. Functions must finish quickly.
- the usage of the underlying Framework should be minimum and wise.
- recursion must be taken into account. It can happen that hooks get called multiple times.
- thread safety must be considered. It can happen that the same hooks get called at the same time by different threads.
- a test must not interfere with other tests. If this happens, the other tests should be stopped for the duration of the last test.

In the end you do not want for the Probe to mess up the results since you are profiling the Target and not the Probe. So high care should be used when dealing with the Probe.

The Qt Probe is implemented in *perfunction.cpp* where the following entry points are defined:

qInspectorActivate(const char *serverName): this function gets called by the Qt Backend to start probe functionality. The Probe activates all the hooks and connects to the server, establishing the communication channel.

qInspectorDeactivate(): this function gets called to stop the Probe, close the communication channel and leave the Target as it was before the injection.

other symbols: there are more symbols present, usually one per Test. An example of this is in section 3.2.2.

The Qt Probe, the Qt Backend and the Inspector tool allow us to move to the next chapter in which we will use those instruments for creating the new, *modern software metrics*.

Chapter 3

Modern Software Metrics

Contents

A *software metric* is a measure of some property of a piece of software. Since quantitative methods have proved so powerful in the all sciences, computer science practitioners and theoreticians have worked hard to bring similar approaches to software development.

In this chapter we will analyze why there is a profound need for metrics, which are the modern requirements and why they are not fulfilled, and develop new methodologies and some useful and intuitive *modern software metrics*.

3.1 Designing software metrics

This dissertation introduces metrics conceived in new, non-ordinary ways. Therefore, before analyzing the metrics, there is the need to introduce the *design process* itself, in terms of *how to think metrics*, *where to start from*, *what problem to address* and *how to produce effective metrics.*

This usefulness of the design process outlined here has been field tested. The results that will be presented in the next sections have proven its goodness.

3.1.1 What metrics are

An agreed definition of "software metric" is: *a measurement of some property of a piece of software.* The definition is loose but really encapsulates all the concepts behind the metric:

the measure: the magnitude of a property of an object, relative to an unit of measurement. A measure can be a number, a scalar field, a vector, any vectors, a measure can be a color, or the triplet of RGB values associated to it or anything else.

the property: an attribute of an object, i.e. the characteristic we are measuring. The property can be a size, a number of elements, the weight of an element, the time taken by some action to happen, and any other thing that can be quantified.

the piece of software: this is what we are analyzing, or comparing. This is the source of the properties we want to measure.

After seeing, in the next section, a brief introduction to the commonly used software metrics and after seeing that they are inadequate the the present and complex software world, we will move to analyzing the requisites of the *modern software metrics.*

3.1.2 Common software measurements

The most used software metrics are:

- bugs per line of code: an a-posteriori measurement of bugs divided by the lines of code.
- code coverage: the degree to which the source code has been tested.
- cohesion: a measure of how focused the various responsibilities of a software module are.
- comment density: how many comments there are over the lines of code.
- coupling: the degree to which each program module relies on other modules.
- cyclomatic complexity: measures the number of linearly independent paths in the source code.
- execution time: the time took for a particular code path to execute.
- function point analysis: an user estimation of the amount of functionality provided by the software.
- instruction path length: the number of machine instructions required to execute a particular code path.
- number of classes and interfaces: the number of classes and interfaces in the source code.
- program load time: the time required to start the program before it gets usable.
- source lines of code: the number of lines of text in the source code.

While widespread and recognized, those metrics are sometimes naive and simplistic. The values are hard to measure (e.g. in *bugs per line of code* that requires foreknowledge), sometimes very subjective (e.g. in *function point analysis*), sometimes not meaningful (e.g. in *comment density*).

As explained in section 1.2 nowadays the software is a lot more complex than when those metrics were introduced, and that is why we are going to introduce new metrics.

3.1.3 Requirements for modern software metrics

Software metrics need to be *useful* and *understandable*. They must *mean* something to the *software analyst*.

The measure itself must be more than just a number, its meaning must be self-evident. The design focus must shift from the process to the user. For a metric to be useful it must be well understood by the software analyst.

So the modern software metrics must reduce the complexity of the problem to something easy to understand and compare. The requirements of the modern software metrics are:

metric intuitiveness: the data gathered must be easy to understand by relating it to familiar concepts. It must not appear out of the blue, but easily correlated with existing objects or concepts. The more the metric is intuitive, the easier is to understand the meaning that stands behind the data and take corrective actions if needed.

presentation intuitiveness: the way the data is presented must carefully designed to be as intuitive as possible. A good metric with bad presentation and analysis tools is worthless since it lacks the tools to express the meaning of the data. The most effective representation is the one the analyst will expect when dealing with the data in the real world.

comparable: the data must be comparable in nature, to highlight difference between two pieces of software or between the same piece of software at different points in time. The presentation, or visualization, must allow such comparisons. This seems to be a fairly simple requirement since numbers can be easily compared, but what must be comparable is the *meaning*, not just the numbers themselves.

The common software measurements, introduced in section 3.1.2 are lacking in regards to the the above requirements. The "comment density" metric, for example, doesn't tell whether a value is better than another, or if there is an optimum value, or how to improve the software, plus it provides no other ways to visualize the metric than treating it as a number of scarce meaning.

3.1.4 The MSM design process

This section outlines the design process to conceive and create the modern software metrics introduced in this dissertation. It is a creative process that involves questioning both the abstract software world and the real world and makes use of Lateral thinking [11].

Question #1. If you could bring a software piece, something very abstract in nature, to the outside world, which properties will it exhibit? Can it blend, burn, melt, turn blue, become bigger, fly away, attract, electrify, spin, emit light or disappear?

Question #2. What is the meaning of doing some operations to the software? Can you apply pressure, transpose it into frequency, find out the step response, measure the temperature (yes, see 3.2), shake it or disassemble it?

Any answer to one of those questions is a candidate for a modern software metric.

The process can start the other way around too, by focusing on a specific problem and finding real world properties that you would like to see or tools that may help you to better expose the problem. The suggestion here is to take the problem or the question that needs an answer (section 1.2 has plenty of them) and try to find a real world equivalent to it.

Example of a design process:

- a piece of software feels slow and this needs to be fixed.
- from the preliminary analysis it appears that the slowness happens somewhere in the graphic system.
- however the user interface is really crowded and it is not even possible to toggle and test the components separately.
- "wouldn't it be nice if the software told me where the problem lies?". The need for a new performance metric arises.

- “wouldn’t it be nice if the graphical system told me where the slowness happens?”. The problem is more circumscribed.
- “could it present it to me as some *temperature* graph, where *red* means *slow*?”. The visualization is conceived.
- “every software should tell you where the performance is lost, with a temperature-like graph”. Section 3.2 is conceived.

Using this design process many new software metrics have been conceived. The next sections will present some of them, their meaning and, where available, their implementation and the results.

3.2 The *Thermal Painting*

One of the most recurring questions for the software analyst is *"why does my software run slow?"* or in a positive formulation *"what can I do to make it faster?"*. *Thermal Painting* is answering this question, at least within certain extents.

Software runs on CPUs that provide the computational power, using energy to perform the needed operations. We can define an *optimized program* as the executable that drains the minimum amount of energy for doing what it is supposed to do. In the same manner we can define an *optimized source code* as the source code that can produce optimized executables.

The *energy metaphor* in software is useful because it gives a fundamental optimization guideline: **to make the program good, or faster, you have to reduce its energy consumption**. Bugs drain energy, inefficient algorithms drain energy, using slower hardware instruction drains energy, but there is another source of energy loss that is "wastes".

Wastes can be everywhere, both in the software that is being developed or in underlying software stacks. Examples of wastes include doing the same operation many times when it is supposed to happen only once (by having unexpected code paths that execute it), or doing some action faster than needed (like drawing some area on the screen faster than the screen refresh rate), or avoiding to free the resources when they are not needed anymore.

One of the major cause of slowdowns in today's software is the painting subsystem. Hardware acceleration is not yet widespread while the users are demanding for good looking and animated user interfaces up to the point to base a buying decision on the performance and appearance of the user interface of the device. This puts a lot of pressure both on the graphical systems and on the applications using them.

3.2.1 Definition

The *Thermal Painting* software metric measures the energy needed to paint a graphical user interface in any configuration, pixel per pixel.

The output of the metric is a scalar two-dimensional matrix. It can be thought of as a grid, with a number of columns equal to the width of the user interface in pixels and the number of rows equal to the height of the user interface in pixels.

The scalar values represent, with a constant scaling factor k, the energy needed by the application to draw the related pixel (that is the user interface pixel associated to the row, col of the matrix).

The scalar values can then be mapped to colors, by associating the blue hue to the minimum value and linearly increasing the hue up to the red, that is associated to the higher value. This way "color maps" can be generated based on the scalar values and superposed to the original GUI image to show the "hot and cold" zones. This software metric was named after this visualization. See figure 3.1 as an example.

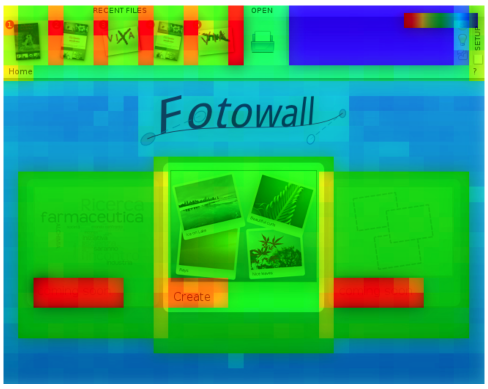


Figure 3.1: Color map example with "hot" and "cold" pixels. The different energy consumption of the pixels is immediately perceived here.

Another way to visualize this metric is to make a two dimensional mesh with the same physical size of the GUI and with a number of vertices equal to the number of the items of the matrix and then map the scalar value at row, col to the z value of the corresponding mesh vertex. In other words, to make a mesh out of the GUI and warp vertices along their z component by a magnitude proportional to the scalar value.

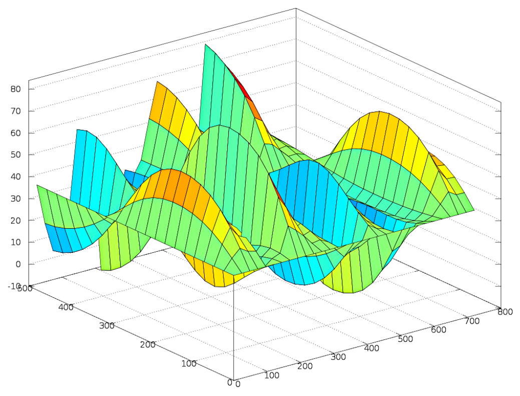


Figure 3.2: Mesh visualization example. A good presentation for the Thermal Painting, especially if the user can interact with the mesh in real-time.

Figure 3.2 shows the three dimensional example, but the results in section 4.2 will show how much this can be extended and how intuitive the data presented in a navigable three dimensional space is.

The next step is to check whether the given definition of *Thermal Painting* is a good definition. The criteria to check the metric have been described in section 3.1.3 and based upon that we can tell that this metric satisfies all the modern requirements: the measure is intuitive because it is associated to the familiar concept that is the temperature so that the hot spots take more energy and the cold take the less, the visualization is intuitive too as either if presented as a color map or a mesh the analyst expects a similar presentation for a temperature measurement, and finally the metric allows the comparison. Different softwares or the same software in different points in time will show a different graph and just doing a per-point subtraction between the matrices (supposed or the same size) will tell which one wins and on which screen areas.

3.2.2 Implementation in Inspector

This section turns into practice the Thermal Painting metric described in the previous section. The metric is implemented as a Backend Module of the Inspector [6] tool, described in chapter 2. There are three components needed: a custom Backend Module, some custom Probe subroutines and a custom presentation/visualization panel.

Since the Qt Backend is available in Inspector, the first implementation of Thermal Painting is done for programs that use the Qt Framework. The rest of this section will refer to some Qt internals, for more information about Qt [10] see the on-line documentation.

The Backend Module

To add some functionality to the Qt Backend, a Backend Module (described in section 2.4.3) must be made. We will call this one *PaintingModule*. The structure of the extension is presented in figure 3.3.

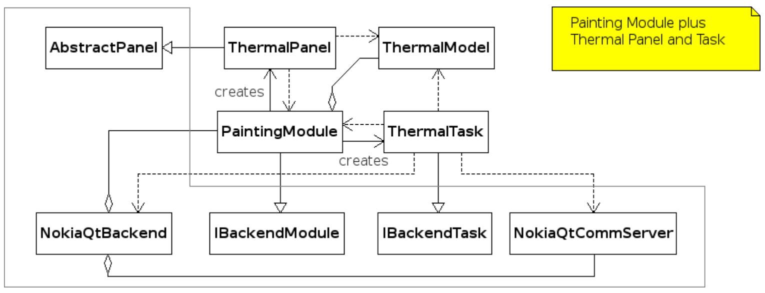


Figure 3.3: The Painting BackendModule for the Qt Backend. The ThermalPanel and the ThermalTask are also presented in this picture.

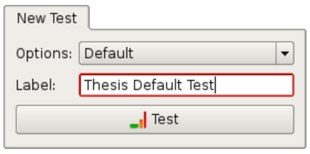
The entry point of the extension is the *PaintingModule*, that has the following relations to the other classes:

1. owns *ThermalModel*, a model containing all the information about previous and present Thermal Painting tests. Such information include the start date, the progress, the completion time estimation, and the actual results (pictures and meshes) of the measurements.

2. creates *ThermalPanel*, the visualization component that allows to display color maps or meshes of all the data in the ThermalModel, interact with them, and load, save, or delete data. This panel is created when Inspector needs one and it is connected to the related ThermalModel through the whole object lifetime.

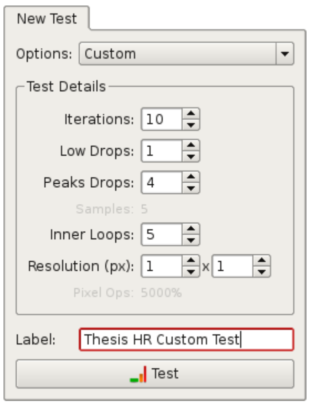
3. creates *ThermalTask*, the component that handles all the communication with the probe, handle the transitions between measure states and store session parameters while implementing the IBackendTask interface too.

The user has some controls over the measurement parameters since there is the change to trade off speed for accuracy in this measure. So at the beginning of a measure, the user selects a preset from the dialog in figure 3.4, or fine tune the parameters as shown on figure 3.5.



Options: Default
Label: Thesis Default Test

Figure 3.4: Configuring the Thermal Painting measure options: the presets dialog.



Options: Custom
Iterations: 10
Low Drops: 1
Peaks Drops: 4
Samples: 5
Inner Loops: 5
Resolution (px): 1 x 1
Pixel Ops: 5000%
Label: Thesis HR Custom Test

Figure 3.5: The advanced options of the Thermal Painting measure.

After the test has been started, the *PaintingModule* creates a *ThermalTask* that asks the probe to perform a new analysis with the given parameters set. When the Probe completes the measurement it sends the results back to the ThermalTask that updates the *ThermalModel* and signals that it (the task) ended. When the ThermalModel is updated with the new data the visualization will show the availability of the data and eventually displays the data. The data on the ThermalModel is preserved between different of Inspector being read on the PaintingModule creation and saved prior its disposition.

The Probing routine

The Probe (see section 2.3 and 2.5.4) is at the heart of the Qt Backend. Living within the program, it allows to do any kind of test and manipulation. In this case it has to do a very precise job: to take the main window of the application and measure the time took by every pixel to be painted.

The Probe subroutine begins here:

```
extern "C"
Q_DECL_EXPORT void qPaintingThermalAnalysis(int passes,
    int headDrops, int tailDrops, int innerLoops,
    int chunkWidth, int chunkHeight, bool consoleDebug)
```

that is the definition of a function with seven parameters identified by a public (exported) symbol name, mangled in C style.

The parameters of the function help to understand how the measurement routine works, so it is better to introduce them first:

passes: this is the number of times the whole window is processed. The higher the value the higher the number of available samples-per-pixel to mediate to find an accurate result.

headDrops: this is the number of samples-per-pixel to discard starting from the highest values. Can help to remove really bad samples due to scheduling or other disruptive delays.

tailDrops: like headDrops but it starts dropping from the lower values.

innerLoops: this is the number of times the function is called in the innermost loop, when effectively measure the time taken by a patch. Can be useful if the measurement time is short compared to the system timer resolution. In this case, multiplying the measures near the measure point can increase the signal-to-noise ratio.

chunkWidth: the width of the patch to measure. If the measure happens pixel-by-pixel both this value and the following will be equal to one.

chunkHeight: the height of the patch to measure.

consoleDebug: a flag that tells the routine to be very verbose and display messages in the *stderr* stream output. This flag is only used by developers.

So, being inside the program, we ask the Qt Framework to redraw a specific patch of 1 square pixels or more and we do this for every pixel, or patch, tesselating the main window of the application. Then all the measures are repeated *passes* times and finally some simple statistical methods are applied to drop invalid measures per-pixel and find the mean value between the "good" measures.

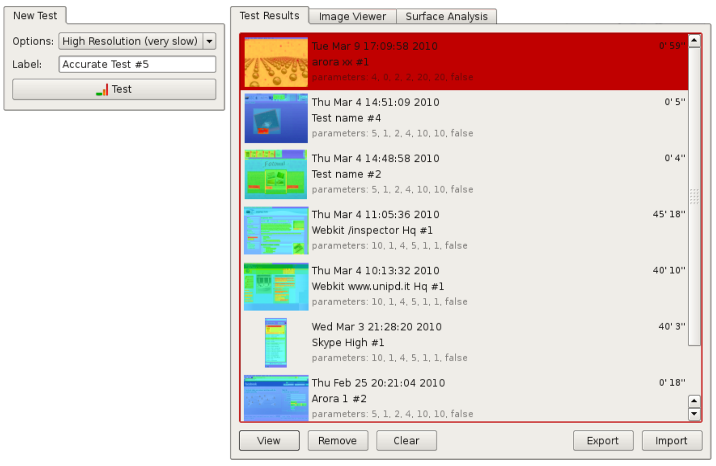
The Probe code, available within the Inspector sources, does exactly what described above, having some nested *for* loops to gather the data and some statistical functions to clean it up plus it regularly transmits a precisely estimated progress value and other state-related information to the Inspector via the socket based communication channel.

When a measure ends, the Probe packs the screen-shot of the window where the analysis was done along as the two dimensional scalar matrix of the results and some meta-data into a binary blob that is sent to Inspector for visualization.

The Visualization

The visualization part of the Thermal Painting consists of three different elements: the panel that can start and control the measurements and shows the list of the results, the color map visualization and the three dimensional visualization.

The Thermal Painting Panel, as shown in figure 3.6 allows to start a new Test (see figure 3.5 too) and to browse the previous ones. The presets combo box contains different presets, from "Fast" to "High Resolution" where the latter is nearly 2500 times slower than the former, and allows the user to set custom values too. On the right side there is a list view showing all the measurements even the ones done previously. On the bottom side there are some controls that allow to *import* or *export* data, *remove* items from the list and completely *clear* the list. By clicking on the *view* button or by double-clicking on an item on the list, the test is loaded into the image viewer.



New Test

Options: High Resolution (very slow)

Label: Accurate Test #5

Test

Test Results | Image Viewer | Surface Analysis

```
Tue Mar 9 17:09:58 2010                     0' 59"
arora xx #1
parameters: 4, 0, 2, 2, 20, 20, false

Thu Mar 4 14:51:09 2010                     0' 5"
Test name #4
parameters: 5, 1, 2, 4, 10, 10, false

Thu Mar 4 14:48:58 2010                     0' 4"
Test name #2
parameters: 5, 1, 2, 4, 10, 10, false

Thu Mar 4 11:05:36 2010                     45' 18"
Webkit /inspector Hq #1
parameters: 10, 1, 4, 5, 1, 1, false

Thu Mar 4 10:13:32 2010                     40' 10"
Webkit www.unipd.it Hq #1
parameters: 10, 1, 4, 5, 1, 1, false

Wed Mar 3 21:28:20 2010                     40' 3"
Skype High #1
parameters: 10, 1, 4, 5, 1, 1, false

Thu Feb 25 20:21:04 2010                    0' 18"
Arora 1 #2
parameters: 5, 1, 2, 4, 10, 10, false
```

View | Remove | Clear | Export | Import

Figure 3.6: The main panel of Thermal Painting. On the right side there are all the data sets.

The Color map visualization, as shown in figure 3.7, just shows the color map of the selected test. On the top right corner it displays a color bar to indicate the full range of the colors. The image is colored mapping linearly the blue tone to the lowest scalar in the result matrix and the red tone to the highest.

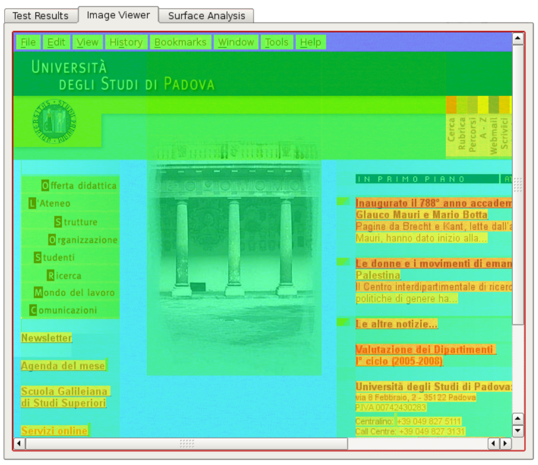
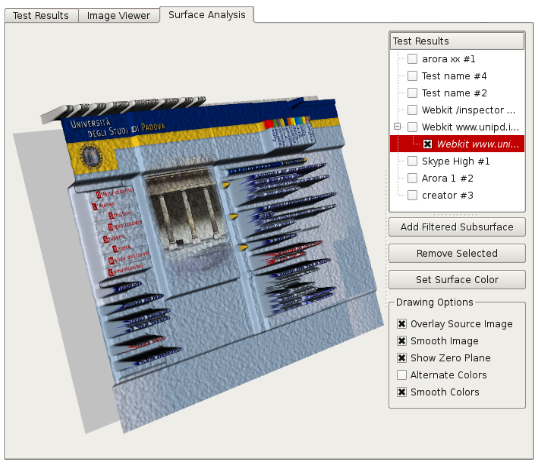


Figure 3.7: The color map display for a website. Energy follows the hue: reddish is hot, meaning high consumption, while blue means low.

The Mesh visualization, as shown in figure 3.8, is really good suited for displaying the results of a Thermal Painting Test. It allows to display one or more Test results at once. In case of a multiple selection, the surfaces become semi transparent so it's easy to understand where they cross and which is the one wasting more energy per-pixel. Being not limited to a small set of values (like the hue in color map visualization) this OpenGL based visualization keeps all the detail of the data so high peaks won't flatten the rest and the detail is fully exposed.



Test Results
- [ ] arora xx #1
- [ ] Test name #4
- [ ] Test name #2
- [ ] Webkit /inspector ...
- [ ] Webkit www.unipd.i...
  - [x] *Webkit www.uni...*
- [ ] Skype High #1
- [ ] Arora 1 #2
- [ ] creator #3

Drawing Options
- [x] Overlay Source Image
- [x] Smooth Image
- [x] Show Zero Plane
- [ ] Alternate Colors
- [x] Smooth Colors

Figure 3.8: The (interactive) mesh display for the same website. The mesh allows a better perception of the magnitude of the energy spent on each pixel.

On the right side there is the Test Results tree and three buttons. The first one, named *Add Filtered Subsurface* creates a child of the selected surface that is a copy of the janitor but convolved with a Gaussian kernel of the given radius. This helps a lot in visually reducing the noise on the figure. The other buttons are used to remove any surface and to set the color of a surface. This option is useful when comparing Test Results by giving each surface a specific color is even easier to understand what's the difference between them. The other options, in form of check boxes, are just to control the rendering: there is the option to use the source image as the texture for the mesh, to use bilinear filtering (an OpenGL interpolation method) on the texture, to show or hide the zero of the energy, the option to use an alternate color scheme and an option to smooth a bit the colors (by interpolating the normals of the surfaces). The visualization pipeline has been build on top of the Vtk [12] toolkit.

The PaintingModule, the Probe routine and the visualizations took nearly 2700 lines of C++ code. Not too much considering that this is a complete implementation of a *modern software metric* for the Qt Framework and it may be highly reusable for other frameworks.

3.2.3 Examples

The Thermal Painting metric was implemented in Inspector and many Tests were conducted to prove its accuracy. Here is a side by side comparison of a test program in two different screen configurations. The Target program is Fotowall [13], a canvas for mixing graphical content.

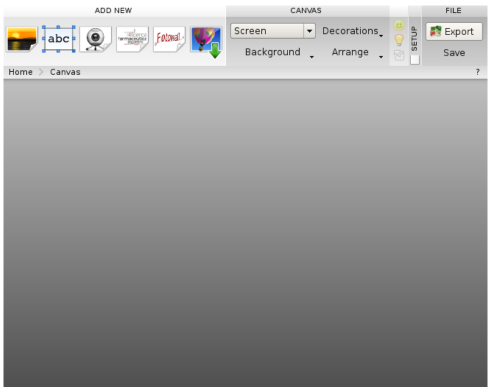
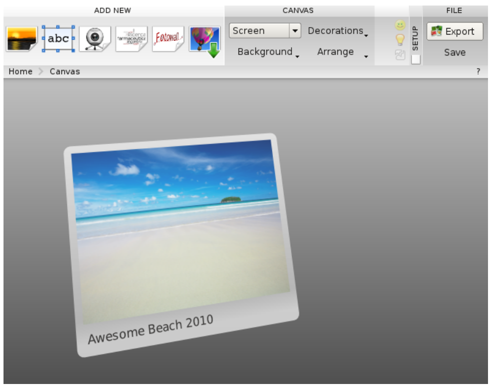
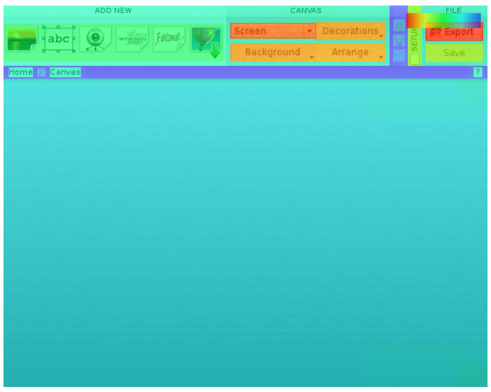
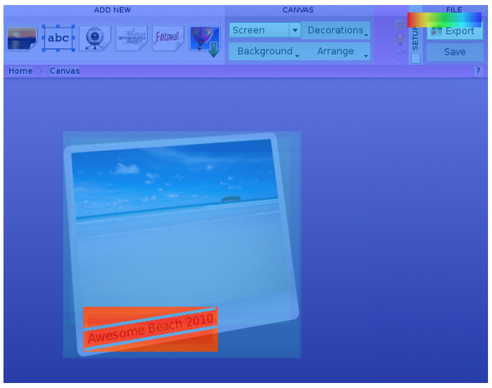
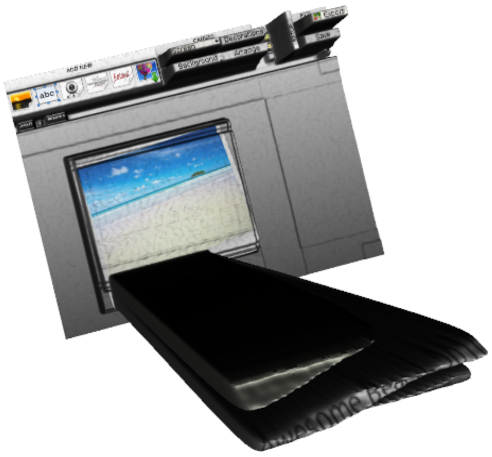
	Canvas empty	With a transformed picture
1.		
2.		
3.		

Table 3.1: Comparison of the Thermal Painting on two screen configurations.

The simple comparison in table 4.2 features an empty canvas on the left side and a canvas with a semi transparent and perspective-transformed picture on the right side. The first row of the table shows the original graphical user interface, the second shows the color-map from the Thermal Painting test and the third one shows a screen capture of the mesh view.

The left column shows the data relative to the measurement over the empty canvas. Here we can see that most of the energy is spent to draw some user controls while, for example, the background gradient on the canvas seems really optimized. The other thing that is visible is the "layered" nature of the energy consumption and this is a direct effect of the user interface being a hierarchy of rectangles where a child is geometrically bounded to its parent. This is why all the children of a parent will take the same *base energy* to be painted, that is the energy consumed by all the ancestors in the recursive pixel painting process. On the right column a picture was added to the canvas, its opacity was set to 0.9 and it was a little perspective transformed. From the pictures we can see that the perspective painting of the picture itself, and of its surrounding frame, does not waste much energy. Instead the painting of the perspective transformed text is really power hungry. In this case we can appreciate the mesh view more than the color map view, since the mesh view doesn't compress the details in case of higher spikes like the color map does. It is more intuitive so more valuable according to the criteria that were discussed in the previous sections.

The energy consumption on transformed text is really high because it uses a completely different pipeline for rendering than the case of the untransformed one, where optimizations such as glyph cache can be applied. Moreover there is another fact that is exposed, and it is probably due to a clipping bug in the Framework: while near the text the power consumption drops down to the expected level, it raises again within the text bounding rectangle. That area should be really clipped out when dealing with the transformed text, as the closer pixels are.

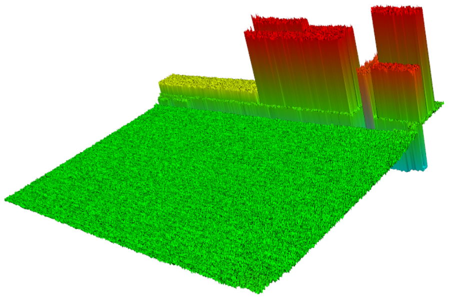
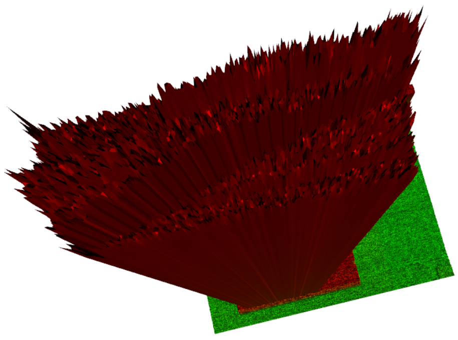
	Canvas empty	**With a transformed picture**
1.		

Table 3.2: Noisy data before the Gaussian filter.

Another interesting thing to see is presented on table 3.2. The table shows the colored mesh view of unfiltered data. After acquiring high resolution data from the Thermal Painting metric it is possible to filter it with a parametric Gaussian kernel to visually smooth the noisy data. The discrete noise is produced by the nature of the measure: it is a measure of time, limited by the resolution of the system timer and by the interferences of external sources such as kernel timers, interrupts, and internal latencies such as CPU, cache or memory ones.

Thanks to the Thermal Painting *modern software metric* everyone has the tools to analyze the performance of a graphical user interface and spot its software or framework problems, and *knowing where the problem lies is a big step towards its resolution.*

Chapter 4

Conclusions

Contents

EVERYBODY makes programming mistakes, but luckily in software we can sometimes build the tools to look for them and fix them. This dissertation introduced the *Inspector* dynamic software analysis tool (on chapter 2) and the *Thermal Painting* modern software metric (on section 3.2) and the design process for creating such metrics (on section 3.1).

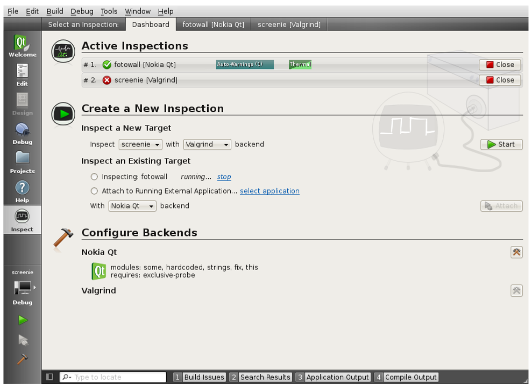
The potential of this new type of software tools is huge and they present an answer to a concrete and growing demand.

4.1 Inspector benefits

The Qt Creator Inspector plug-in, that is currently at version 1.3.82, is very concrete and has a solid behavior. The most prominent Inspector features are:

- offers the complete Thermal Painting software metric.
- offers some other minor metrics and tools, not described here.
- it allows to do a wide range of operations over existing and even already running programs, without requiring any source or binary modifications.
- it is the ideal software platform for developing modern software metrics.
- it is contributed as open sourced under the LGPL 2.1 license so that everyone is free to use it an contribute to it.

The dashboard of Inspector, shown in figure 4.1, is very easy to use and allows to quickly start inspections over new or running Targets and to monitor what is going on in all the active inspections from a central place.



Active Inspections

1. fotowall [Nokia Qt]

2. screenie [Valgrind]

Create a New Inspection

Inspect a New Target

Inspect screenie with Valgrind backend

Inspect an Existing Target

Inspecting: fotowall *running...* stop

Attach to Running External Application... select application

With Nokia Qt backend

Configure Backends

Nokia Qt

modules: some, hardcoded, strings, fix, this
requires: exclusive-probe

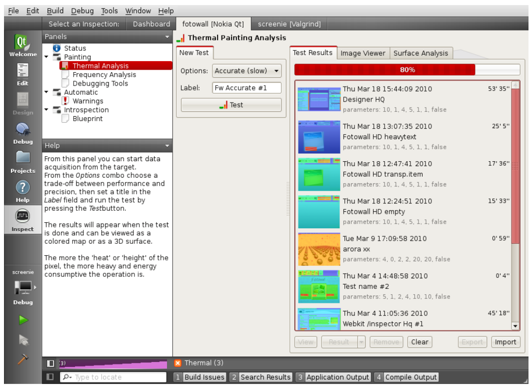
Valgrind

Figure 4.1: The dashboard of Inspector 1.3.82. This panel allows to start new Inspections and gives a quick overview over the existing ones.

The Inspection window, shown in figure 4.2, allows to choose an available tool from the left sidebar and operate with that tool on the right side of the window. Context sensitive help is provided on the left sidebar and in every moment there is a scrolling list of the active tasks on the bottom of the window and the possibility to cancel the tasks (if interruptible) too.

4.1.1 Open challenges

While Inspector has just begun showing its potential and works good for the common use cases, there already are some issues that need to be addressed. While



From this panel you can start data acquisition from the target.
From the *Options* combo choose a trade-off between performance and precision, then set a title in the *Label* field and run the test by pressing the *Test*button.

The results will appear when the test is done and can be viewed as a colored map or as a 3D surface.

The more the 'heat' or 'height' of the pixel, the more heavy and energy consumptive the operation is.

Options: Accurate (slow)
Label: Fw Accurate #1

80%

```
Thu Mar 18 15:44:09 2010                53' 35"
Designer HQ
parameters: 10, 1, 4, 5, 1, 1, false
Thu Mar 18 13:07:35 2010                25' 5"
Fotowall HD heavytext
parameters: 10, 1, 4, 5, 1, 1, false
Thu Mar 18 12:47:41 2010                17' 36"
Fotowall HD transp.item
parameters: 10, 1, 4, 5, 1, 1, false
Thu Mar 18 12:24:51 2010                15' 33"
Fotowall HD empty
parameters: 10, 1, 4, 5, 1, 1, false
Tue Mar 9 17:09:58 2010                 0' 59"
arora xx
parameters: 4, 0, 2, 2, 20, 20, false
Thu Mar 4 14:48:58 2010                 0' 4"
Test name #2
parameters: 5, 1, 2, 4, 10, 10, false
Thu Mar 4 11:05:36 2010                 45' 18"
Webkit /inspector Hq #1
```

Figure 4.2: A running Inspection with a Thermal Painting measure running. Notice the good amount of feedback about the operations being done.

some of them are just simple missing features, some others are real challenges, like:

multi-threading support: right now the Probe is made with static hooks that are called at some point in time and may be called by different threads. If that happens, a per-thread storage model should be used and the measured data should be grouped and transmitted per-thread. Of course thread-safety of the shared data will be required at that point while for now it is not granted. The first component that will be affected by threading issues is the event loop monitor, partially implemented in the Probe and used by Inspector, but not mentioned in this dissertation.

more precise timing: tasks such as a Thermal Painting measure need precise timing information about the real run time (or cpu cycles spent) per-process or per-thread. Right now Inspector is using the standard POSIX timing functions, with micro-second accuracy and is improving the information through statistical means. Various other approaches including the *setitimer* and the *taskstat* Linux^{TM}kernel accounting interface have been tested but the results were less precise than with the standard timing interface. To reduce the jittering due to the kernel scheduling and avoid interferences from other processes that have to run at the same time some critical timing function is running, it is possible to set the CPU affinity of a running executable to a single CPU (so that the task is not transferred between CPUs), lower the niceness of the Target process (meaning giving it higher priority) and if possible schedule the process as runtime. On Linux this can be accomplished with the `schedtool` command.

safe probe injection: right now the probe injection happens in a casual point in time that is near the startup of the program (if the Target is run in Inspector) or during its execution (if Inspector has attached to the Target during its execution). However at that time the Target process can be in any state and it is not guaranteed that calling the probe activation function will not interfere with the Target or crash it. A solution could be to just call thread-safe functions when calling the probe activation function and schedule the more critical operations for later execution, either in a "phase two" call to the probe or by adding code to be executed by the right event loop at the right point in time.

allow for dynamic code modifications: Inspector Backends make good use of the knowledge they have about the Framework they are dealing with, and this allows to use static public symbols, (un)documented internal hooks, to integrate with the event loop and a lot of other jobs. However having the possibility to alter the Target code, placing callbacks dynamically would allow for even more functions. For example callbacks could be set on the add and remove functions of the lists (to check for overgrowing lists, an usual source of problems), or on the send and receive functions of network sockets. This is a well documented research topic.

standardize the probe communication protocol: right now the Probe is using a stream protocol to transmit the data to Inspector. The protocol of the data on the wire has not been standardized and this means that if somebody wants to implement a new probe-oriented Inspector Backend, he has to implement the Probe and potentially duplicate lots of the software stack of another Inspector Backend. However, if the protocol was standardized, then adding the support for a new Framework could mean to just rewrite the Probe for that framework while reusing some existing Inspector software stacks for the data analysis and visualization. An great example of this could be to add a *gtk+* probe that transmits data to Inspector to perform the Thermal Painting analysis on that Framework too.

4.1.2 Social impact

At the time of writing, Inspector has not been public announced and it has only been shown to a small group of people. However there is a lot of interest for both the new tool and the metrics that will come with it.

Inspector will allow to compare different softwares from new points of view. It will allow for example to compare two similar programs for a cellphone and see which one performs better, or which one drains less energy from the battery. There are software testing suites and continuous integrations systems that may include the Inspector tests in their test sets, since this will allow to immediately spot performance gains or losses in many areas.

The modern software metrics implemented over Inspector could also be used as a standard for software evaluation or certification. They could be used for example for certifying that a software is "green", or that it does not drain power when idle, or that it does it not crash when given a billion of random stimuli.

4.2 The Thermal Painting metric

Even if not yet public, the Thermal Painting has already been used to fix performance problems in some programs. The most notable example is the Qt Creator itself where a painting problem in the text editor component, being slower to paint as far as the line number increased, was identified and fixed.

Other relevant features of the Thermal Painting metric and visualization are shown in the next figures.

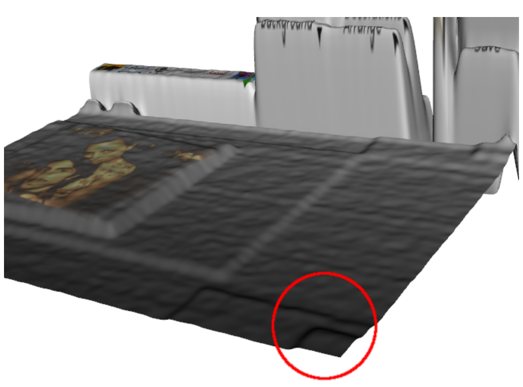


Figure 4.3.

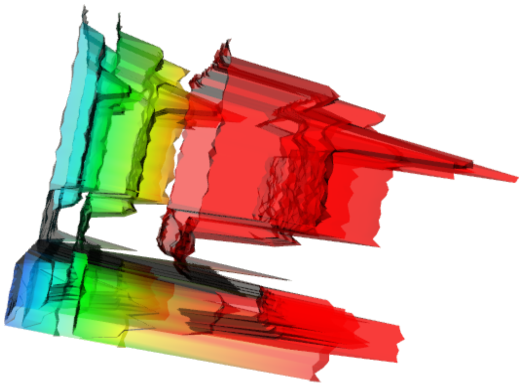


Figure 4.4.

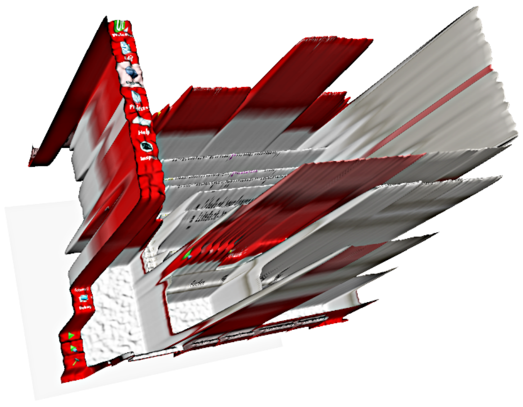


Figure 4.5.

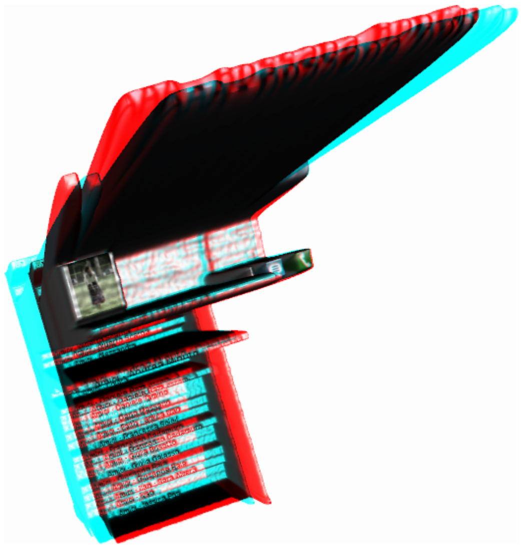


Figure 4.6.

Figure 4.3 shows a kind of "step" in the corner of the mesh. In the canvas there is no visible element located in that corner, but hovering the corner with the mouse it shows a small icon right there. Even if the icon is hidden, the painting functions recurse into that component and this calls take some time. When filtering the data this step, and other similar steps, becomes evident even if they last for just some fraction of a microsecond.

In figure 4.4, three semi transparent colored meshes are shown. When more than a mesh is selected in the mesh view component they are made semi transparent for allowing to visually evaluate the volume of the difference. In this case the meshes are just displaced but even better visual results can be obtained with intersecting meshes.

A Thermal Painting measure over the quite complex QtCreator graphical user interface is shown in figure 4.5. The only difference in profiling a complex graphical user interface than profiling a simple and small interface is that more time is needed to perform the measure over the complex user interface. This is a direct consequence of the direct proportionality between the energy consumption of the graphical user interface and the time needed to perform the operation. However a future development could be to random sample the user interface at the beginning and auto-tune inner loops for speeding up the process while maintaining a good precision over the measures.

The last figure, 4.6, shows an anaglyph red-cyan rendering of a Thermal Painting mesh. This measurement is done over the well known and proprietary Skype program, to prove that the Thermal Painting metric needs no modifications to the source code. Other ways for viewing the mesh in 3D are provided, like: red-blue anaglyph, horizontal interlace, vertical interlace, pure left, pure right, and checkerboard.

4.2.1 Social impact

The Thermal Painting software metric is expected to have a considerable social impact. At the time of writing there are tens of thousands of programs written against the Qt Framework and a conservative estimation is that at least a tenth of those can be profiled using this metric. As an additional vehicle, the open source world, is really permeable to this kind of innovation and there are hundreds of persons just waiting for tools like Inspector and for metrics like the

Thermal Painting to show up to optimize their programs.

Given those estimations, there are thousand of programs that can be quickly profiled and fixed, in addition to the Qt Framework fixes that will allow everybody using Qt to immediately experience the performance improvements.

Given the world scale of just the open source movement and the fast speed at which changes propagate nowadays, it is fairly possible that the energy savings on a world scale will quick be relevant.

4.3 Future directions

This was just the beginning of the work in this field. Inspector is a proven tool that will help developers to profile, compare, and improve their code in new ways.

There will be new Backends added, and the next candidates are the Valgrind Backend (that allows memory checking and function calls and stack profiling) and the OpenGL Backend (that with his probe will give access to the OpenGL state, e.g. textures).

New metrics will be added to the Qt Backend too, and the first ones will be: the *Heartbeat* (a way to visually show what a program is really doing while it seems idle), the *Painting Frequency* (finding out the screen refresh rate per-pixel) and a *Testing Module* that will allow recording user input, sending random input, and automating the application via scripting.

In the spirit of openness everyone is invited to contribute.

Bibliography

[1] Geoff Tennant. *Six Sigma : SPC and TQM in Manufacturing and Services.* Ashgate Publishing, 2001.

[2] Wikipedia. Application binary interface. http://en.wikipedia.org/wiki/application_binary_interface.

[3] Google Chrome. http://www.google.com/chrome.

[4] The WebKit Open Source Project. http://webkit.org.

[5] NVIDIA Parallel Nsight. http://developer.nvidia.com/object/nexus.html.

[6] Inspector. http://www.enricoros.com/opensource/inspector.

[7] Valgrind. http://www.valgrind.org.

[8] Factory Pattern. http://en.wikipedia.org/wiki/factory_pattern.

[9] Qt Creator: Cross-Platform Qt IDE. http://qt.nokia.com/products/developer-tools.

[10] Qt: A cross-platform application and UI framework. http://qt.nokia.com.

[11] Edward De Bono. *Lateral Thinking - Creativity Step by Step.* Perennial Library, 1970.

[12] Visualization Toolkit.
http://www.vtk.org.

[13] Fotowall.
http://www.enricoros.com/opensource/fotowall.