Constant Growth Software Development

by Jan Hakenberg based on discussions with Claudio Ruch
2019-10-29

Abstract

We present a systematic of programming activities and design principles that we consider to be necessary for steady, long-term growth of a software project. Every aspect of the software development has to scale with the size of the code base. Compromising on the principles reduces effectiveness of the developers.

Collaborating on a large software project is a social effort: Developers base their work on previous contributions and in turn create new functionality for themselves and future users to rely on. Writing tests and comments, may benefit future developers more than the programmer and therefore are altruistic acts. At all times, the overall software architecture and style has to inspire the developers to produce quality code in turn.

The allocation of work balances different programming activities, for instance feature development vs. maintenance. The programming language and tools must be suitable for the entire spectrum of tasks. We describe how traditional, time-consuming pitfalls can be avoided.

Using the go-kart as a use case, we illustrate how abstract design concepts were applied in practice, but also point to known shortcomings in our development pipeline.

Programming Activities

	Topics	**Summary**
New Features	vision/ideas research project specifications rapid prototyping	The objective is to create a working implementation fast. The other activities will help to provide a neat and tidy environment to try-out/implement new ideas. Once new functionality has proven to be useful, the evaluation and repair work begins.
Visualization	post-processing analysis reporting graphical user interface runtime efficiency	If the software processes a lot of data, then visualization of the data/intermediate-results allows to detect anomalies, and to be inspired for improvements/next-steps.
Maintenance	architecture design conventions refactoring/beautification abstraction/eliminate redundancy modularization generalization/library	Maintenance results in many small files, functions with few lines of code, immutable objects, neat dependency trees/library layers, reduced visibility, and uniform code format. Efforts often reduce the line count of the code base, and generalize code from the application layer to a library.
Testing	correctness/verification documentation code statistics end-user/running applications feedback	Rule of thumb: Every freshly implemented, math-heavy function in the main scope contains three mistakes on average. The mistakes may be identified by writing tests. Tests fortify the implementation, and motivate to write more precise documentation, for instance the behavior of a function on invalid input.

Types of Contributors

	Developers: New Features/Visualization	**Reviewers: Maintenance/Testing**
Enjoy	adding new features	rewriting code, writing tests
Need	slim/well-documented interfaces	immutability, reduced visibility, license to rename variables/functions, and relocate functions
Awareness	≤ 15% of all functionality	close to 100% of the code base
Suffer from	bugs in the library layer	large files/functions, duplicate code, classes with many fields
Minimal horizon	3 months	1 year

Conquer the Divide

Design Pitfalls	**Resolution**
un-/signed int/long, float/double	a single abstract scalar type
vector, matrices	unifying concept: tensor
many lcm/ros/etc. messages	a single omnipotent message type: variable-length binary array
several languages, scripts, makefiles	use only one programming language with a simple build command
compile time increases with size of code base	low, constant amortized compile time

Timeless Pitfalls	**Resolution**
duplicate code with minor differences	object-oriented design
interwoven utility and application code	distribute functionality across libraries
incorrect implementation of mathematical functions	correct implementation of mathematical functions

Unqualified Languages

Cpp: slow compilation, no introspection, complicated dependency management, compilers don’t agree on definition of language:
https://www.youtube.com/watch?v=tsG95Y-C14k

Python: no immutable fields, no strictly private functions, does not encourage object-oriented programming
https://mail.python.org/pipermail/tutor/2003-October/025932.html

Because Cpp (IDEs do not parse the code reliably) or python (language allows too much) do not allow effective reviewing and refactoring, the cost of maintenance grows exponential in the size of the project.

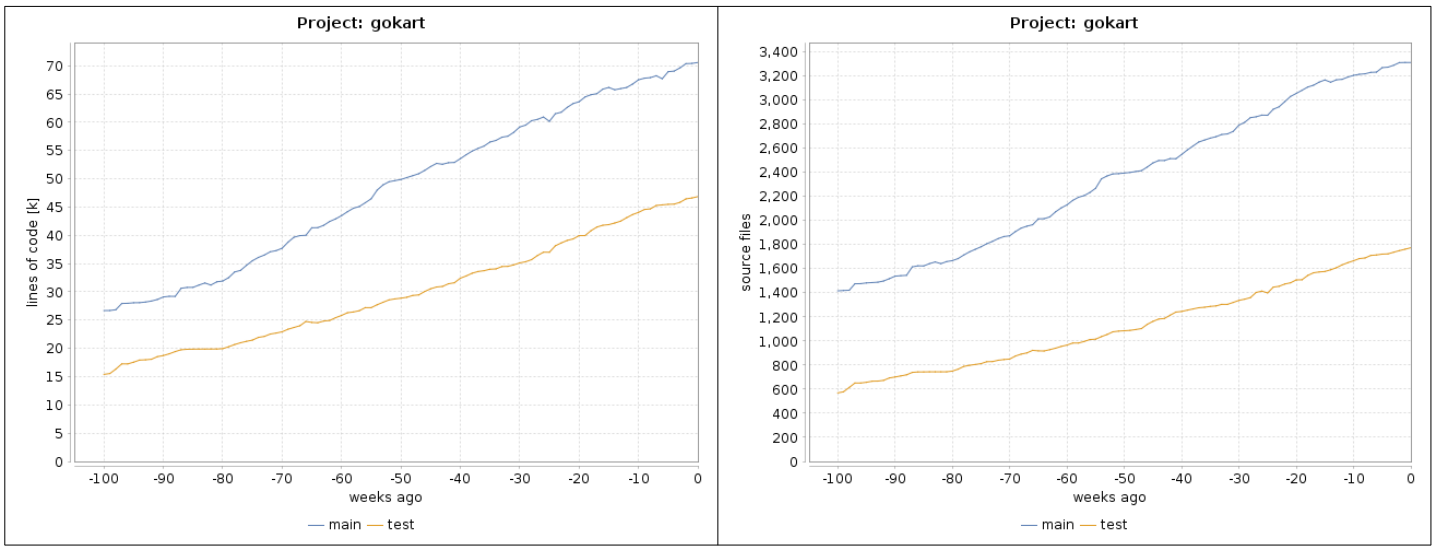
Guidelines applied in the Go-kart Project

Concept	**Examples of Implementation**
Immutable objects	All sensor/actuator messages are read-only. The keyword *final* is used 8149 times in the code base.
Reduced visibility	The keyword *private* is used 7142 times in the code base.
Minimal redundancy	Implemented in a single place and used everywhere: Dot product, IIR1 filter, ...
Introspection	Appending a human-configurable parameter is a one-line change. The GUI, and parameter storage is managed automatically through introspection
Object-oriented design	*ManualControlInterface* is an interface implemented by a joystick configuration, as well as the ADC readout. Switching between the two operation modes is a one-line change.
Abstraction	The generalization of affine combinations to points from non-linear spaces
Test code	40% of all code are tests
Respect the hardware	Carrying out experiments with the robot is the most time consuming activity. Any preparatory measure that prevents the need to redo experiments is well invested.

Shortcomings of the Go-kart Project

Circumstance	**Negative implications**
Dependency on commercial numerical solver	The commercial third party solver is not available in the continuous integration. Safety critical parts of the software are not covered by tests that run everywhere.
Reluctancy to investigate characteristics of hardware components	The reliance on hardware mandates an thorough investigation of the properties of the component; however, unless when anomalies occur, we do not allocate more resources beyond reaching a working version.
Desire to develop features; few incentives to write tests/documentation	Not all code in the application layer is reviewed immediately. In consequence, a mistake may go unnoticed for a while. Thorough review is reserved predominantly for core parts.
Frequent interruptions, hidden agendas of the project management, and a management that is ignorant of the engineering reality	These unfavorable distractions from the engineering work result in a reduction of effectiveness and technical quality of the project. There are also psychological effects.

Statistics of the Go-kart Project



References

T. DeMarco, T. Lister: Peopleware; Productive Projects and Teams
T. Schulz: Was Google wirklich will