What is SObjectizer-5.7?

(at version 5.7.0)

SObjectizer Team, Jan 2020

SObjectizer is a framework for building robust multithreaded applications.

It is based on async message exchange and uses a mixture of high-level abstractions:

- Actor-Model
- Publish-Subscribe
- Communicating Sequential Processes

SObjectizer's main ideas and principles were formulated in the middle of 1990s, during the development of SCADA Objectizer project in Development Bureau of System Programming in Homyel, Belarus (1996-2000).

SCADA Objectizer's ideas were reused in the new project SObjectizer-4 in 2002.

Evolution of SObjectizer-4 was stopped in 2010 and the development of SObjectizer-5 started.

SObjectizer was an in-house project of *Intervale*[*] for the long time.

Since 2013 SObjectizer is an independent project which a totally separated from *Intervale*.

Since 2016 the SObjectizer's development and support is performed by ***stiff****stream*[**].

[*] www.intervale.ru
[**] stiffstream.com

SObjectizer was used for:

- SMS/USSD traffic service;
- financial transaction handling;
- software parameters monitoring;
- automatic control of the theatre's scenery*;
- machine learning;
- prototyping of distributed data-acquisition software;
- components of DMP in an advertising platform;
- components of an online game.

*https://habr.com/en/post/452464/

SObjectizer can be used for the development of a large, distributed and highly loaded software systems.

SObjectizer can also be used for small utilities and quick-and-dirty prototypes.

The whole application can be built upon SObjectizer.

Or SObjectizer can be used only for a small part of an application developed on the top of Qt, wxWidgets, ACE, Boost, etc.

What distinguishes SObjectizer?

Maturity

SObjectizer is based on ideas that have been put forward in 1995-2000.

And SObjectizer itself is being developed since 2002.

SObjectizer-5 is continuously evolved since 2010.

Stability

From the very beginning SObjectizer was used for business-critical applications, and some of them are still being used in production.

Breaking changes in SObjectizer are rare and we approach to them very carefully.

For example, branch 5.5 respects compatibility for more that four years.

Cross-platform

SObjectizer runs on Windows, Linux, FreeBSD, macOS and Android.

Easy-to-use

SObjectizer provides easy to understand and easy to use API with a lot of examples in the SObjectizer's distributive and a plenty of information in the project's Wiki*.

https://github.com/Stiffstream/sobjectizer/wiki

Free

SObjectizer is distributed under BSD-3-CLAUSE license, so it can be used in development of proprietary commercial software for free.

It's not dead!

SObjectizer is one of a few live and evolving OpenSource actor frameworks for C++.

There are more that 30 releases on SObjectizer-5 since it was open-sourced in May 2013.

It evolves

SObjectizer-5.7 has much more features than the first public release of SObjectizer-5 had in 2013.

During the evolution, SObjectizer incorporated several features that can be called game-changers...

Some of new features since 2013:

- agents as hierarchical state machines;
- mutability for messages;
- message chains;
- environment infrastructures;
- enveloped messages;
- dead-letter handlers;
- message-tracing;
- stop-guards;
- run-time monitoring;
- unit-testing of agents;
- ...

Let's dig into some details!

Using SObjectizer-5.7 a programmer must define messages/signals and implement agents for processing them.

Agents are created by the programmer and are bound to dispatchers. Dispatchers are responsible for message dispatching and providing of working thread on which agents handle messages.

A programmer can create as many agents as needed.

Agent is a lightweight entity.

There could be thousands, millions and hundreds of millions of agents.

Number of agents is limited only by amount of RAM and the common sense of a programmer.

A traditional "Hello, World" example. This is the main agent:

```
#include <so_5/all.hpp>

class hello_actor final : public so_5::agent_t {
public:
	using so_5::agent_t::agent_t;

	void so_evt_start() override {
		std::cout << "Hello, World!" << std::endl;
		// Finish work of example.
		so_deregister_agent_coop_normally();
	}
};
```

And this is the main() function:

```
int main() {
	// Launch SObjectizer.
	so_5::launch([](so_5::environment_t & env) {
		// Add a hello_actor instance in a new cooperation.
		env.register_agent_as_coop( env.make_agent<hello_actor>() );
	});

	return 0;
}
```

Let's see an example of timers and Publish/Subscribe.

There will be a producer agent that will distribute new values on a periodic basis.

And there will be a couple of consumers on that data.

Define a message with new data:

```
#include <so_5/all.hpp>

using namespace std::literals;

// This message will be published to a multi-consumer message box on a periodic basis.
struct acquired_value {
	std::chrono::steady_clock::time_point acquired_at_;
	int value_;
};
```

Agent-producer (fields and the constructor):

```
class producer final : public so_5::agent_t {
	const so_5::mbox_t board_; // The destination for acquired_value.
	so_5::timer_id_t timer_;
	int counter_{};

	// This signal will be sent by the timer.
	struct acquisition_time final : public so_5::signal_t {};

public:
	producer(context_t ctx, so_5::mbox_t board)
		:  so_5::agent_t{std::move(ctx)}, board_{std::move(board)}
	{}
```

Agent-producer (the main methods):

```
        void so_define_agent() override {
                // A subscription to periodic signal.
                so_subscribe_self().event( [this](mhood_t<acquisition_time>) {
                        // Publish the next value for all consumers.
                        so_5::send<acquired_value>(
                                board_, std::chrono::steady_clock::now(), ++counter_);
                } );
        }

        void so_evt_start() override {
                // Agent will periodically recive acquisition_time signal
                // without initial delay and with period of 750ms.
                timer_ = so_5::send_periodic<acquisition_time>(*this, 0ms, 750ms);
        }
};
```

Agent-consumer:

```
class consumer final : public so_5::agent_t {
	const so_5::mbox_t board_;
	const std::string name_;

	void on_value(mhood_t<acquired_value> cmd) {
		std::cout << name_ << ": " << cmd->value_ << std::endl;
	}

public:
	consumer(context_t ctx, so_5::mbox_t board, std::string name)
		: so_5::agent_t{std::move(ctx)}, board_{std::move(board)}, name_{std::move(name)}
	{}

	void so_define_agent() override { so_subscribe(board_).event(&consumer::on_value); }
};
```

The main() function:

```
int main() {
	so_5::launch([](so_5::environment_t & env) {
		auto board = env.create_mbox();
		env.introduce_coop([board](so_5::coop_t & coop) {
			coop.make_agent<producer>(board);
			coop.make_agent<consumer>(board, "first"s);
			coop.make_agent<consumer>(board, "second"s);
		});

		std::this_thread::sleep_for(4s);
		env.stop();
	});
	return 0;
}
```

All agents in the example above works on the same default dispatcher.

Let's bind them to different dispatchers...

```
so_5::launch([](so_5::environment_t & env) {
	auto board = env.create_mbox();
	env.introduce_coop([&](so_5::coop_t & coop) {
		// A separate dispatcher for producer.
		coop.make_agent_with_binder<producer>(
			so_5::disp::one_thread::make_dispatcher(env).binder(), board);

		// And a separate dispatcher for consumers.
		auto disp = so_5::disp::active_obj::make_dispatcher(env);
		coop.make_agent_with_binder<consumer>(disp.binder(), board, "first"s);
		coop.make_agent_with_binder<consumer>(disp.binder(), board, "second"s);
	});
	...
});
```

SObjectizer has several ready-to-use dispatchers:

- ***one_thread***. All agents work on a single working thread;
- ***active_obj***. Every agent works on a single dedicated working thread;
- ***active_group***. A single dedicated working thread is allocated for a group of agents;
- ***thread_pool***. A working thread is selected from thread pool. Agents can be moved from one working thread to another. But an agent can't work on two threads at the same time;
- ***adv_thread_pool***. A working thread is selected from thread pool. Agents can be moved from one working thread to another. Moreover an agent can work on several threads at the same time (if the agent's event handlers are marked as thread safe);
- ***prio_one_thread*** (***strictly_ordered*** and ***quoted_round_robin***). One working thread and dispatching with respect to agent's priorities;
- ***prio_dedicated_threads::one_per_prio***. One working thread per a priority.

A programmer can create any number of dispatchers needed. This allows to bind agents to different context in such a way that the impact of one agent on another will be minimal. For example:

- *one one_thread dispatcher for AMQP-client agent;*
- *one thread_pool dispatcher for handling requests from AMQP-queues;*
- *one active_obj dispatcher for DBMS-related agents;*
- *yet another active_obj dispatcher for agents whose work with HSMs connected to the computer;*
- *and yet another thread_pool dispatcher for agents for managing all the stuff described above.*

SObjectizer even allows writing an application without actors...

You can write a multithreaded application using only plain std::thread and SObjectizer's mchains.

mchain in SObjectizer is an analog of CSP-channel.

Let's see a ping-pong between worker threads via mchains...

Ping-pong on plain threads and mchains (1)

```
#include <so_5/all.hpp>

struct ping {
	int counter_;
};

struct pong {
	int counter_;
};
```

Ping-pong on plain threads and mchains (2)

```
void pinger_proc(so_5::mchain_t self_ch, so_5::mchain_t ping_ch) {
	so_5::send<ping>(ping_ch, 1000); // The initial "ping".

	// Read all message until channel will be closed.
	so_5::receive( so_5::from(self_ch).handle_all(),
		[&](so_5::mhood_t<pong> cmd) {
			if(cmd->counter_ > 0)
				so_5::send<ping>(ping_ch, cmd->counter_ - 1);
			else {
				// Channels have to be closed to break `receive` calls.
				so_5::close_drop_content(self_ch);
				so_5::close_drop_content(ping_ch);
			}
		});
}
```

Ping-pong on plain threads and mchains (3)

```
void ponger_proc(so_5::mchain_t self_ch, so_5::mchain_t pong_ch) {
	int pings_received{};

	// Read all message until channel will be closed.
	so_5::receive( so_5::from(self_ch).handle_all(),
		[&](so_5::mhood_t<ping> cmd) {
			++pings_received;
			so_5::send<pong>(pong_ch, cmd->counter_);
		});

	std::cout << "pings received: " << pings_received << std::endl;
}
```

Ping-pong on plain threads and mchains (4)

```
int main() {
	so_5::wrapped_env_t sobj;

	auto pinger_ch = so_5::create_mchain(sobj);
	auto ponger_ch = so_5::create_mchain(sobj);

	std::thread pinger{pinger_proc, pinger_ch, ponger_ch};
	std::thread ponger{ponger_proc, ponger_ch, pinger_ch};

	ponger.join();
	pinger.join();

	return 0;
}
```

SObjectizer-5.7 provides a select() function that can be compared with Go's select statement.

SObjectizer's select() allows doing non-blocking send to a mchain with waiting of an incoming message from another mchain(s).

Let's see how famous Go's example* with the calculation of Fibonacci numbers in different goroutines can look like in SObjectizer.

https://tour.golang.org/concurrency/5

Calculation of Fibonacci numbers in different threads (1)

```
#include <so_5/all.hpp>

#include <chrono>

using namespace std;
using namespace std::chrono_literals;
using namespace so_5;

struct quit {};
```

Calculation of Fibonacci numbers in different threads (2)

```
void fibonacci( mchain_t values_ch, mchain_t quit_ch ) {
  int x = 0, y = 1;
  mchain_select_result_t r;
  do {
    r = select( from_all().handle_n(1),
      send_case( values_ch, message_holder_t<int>::make(x),
          [&x, &y] {
            auto old_x = x;
            x = y; y = old_x + y;
          } ),
      receive_case( quit_ch, [](quit){} ) );
  } while( r.was_sent() && !r.was_handled() ); // Continue while nothing received.
}
```

Calculation of Fibonacci numbers in different threads (3)

```
int main() {
  wrapped_env_t sobj;

  thread fibonacci_thr;
  auto thr_joiner = auto_join( fibonacci_thr ); // Automatically join thread at exit.

  auto values_ch = create_mchain( sobj, 1s, 1, // Limit the capacity of Fibonacci numbers chain.
        mchain_props::memory_usage_t::preallocated, mchain_props::overflow_reaction_t::abort_app );
  auto quit_ch = create_mchain( sobj );

  fibonacci_thr = thread{ fibonacci, values_ch, quit_ch }; // A separate thread for the calculation.

  receive( from( values_ch ).handle_n( 10 ), []( int v ) { cout << v << endl; } );

  send< quit >( quit_ch );
}
```

SObjectizer is not a silver bullet

SObjectizer supports several concurrency models.

SObjectizer makes the development of complex multithreaded applications easier.

It's proved by 18 years of usage "in production".

But SObjectizer doesn't solve all the problems...

SObjectizer is responsible for:

- in-process message dispatching;
- providing working thread for message processing;
- SObjectizer Run-Time's parameters tuning;
- collecting of run-time stats (if enabled);
- message delivery process tracing (if enabled).

And SObjectizer-5 doesn't support development of distributed application just "out of box".

Additional libraries and tools should be used for that.

Some more info about SObjectizer-5.7.0

SObjectizer Team, Jan 2020

SObjectizer-5.7 is developed using C++17.

Supported compilers and platforms:

- Visual Studio 2019, GCC 7.1-9.2, clang 6.0-9.0
- Windows, Linux, FreeBSD and MacOS.

Support of other platforms is possible in the case when SObjectizer's developers will have an access to those platforms.

Version 5.7.0 is ~31 KLOC of SObjectizer core.

Plus ~41 KLOC of tests.

Plus ~8 KLOC of samples.

Plus SObjectizer's core documentation*.
Plus articles and presentations** (some of them in Russian).

*https://github.com/Stiffstream/sobjectizer/wiki
**http://sourceforge.net/p/sobjectizer/wiki/Articles/

Useful references:

Project's home: https://github.com/Stiffstream/sobjectizer

Documentation: https://github.com/Stiffstream/sobjectizer/wiki

Google-group: https://groups.google.com/forum/#!forum/sobjectizer

Support: https://stiffstream.com