The *GenomeTools* Developer's Guide

Sascha Steinbiss, Gordon Gremme and Stefan Kurtz*

February 4, 2013

Contents

1 Introduction

This document describes design properties and coding guidelines for the *GenomeTools* genome analysis system. The goal of the *GenomeTools* environment is to provide a well understandable, comprehensive and most importantly reusable set of classes and modules to aid in the development of C-based bioinformatics applications.

The expected gain in productivity is only possible to achieve if all components of the *GenomeTools* behave in a similar way – or so to say – in a way which is least surprising to the user (which is, in this case, a programmer). Thus we ask all developers contributing code to the *GenomeTools* to adhere to a common set of rules which make it easier for others to reuse the products of everyone's hard work.

*please send comments to: `steinbiss@zbh.uni-hamburg.de`

2 Object-oriented design

2.1 Classes

The central component type in *GenomeTools* is the *class*. Structuring the C code into classes and modules gives us a unified design approach which simplifies thinking about design issues and avoids the code base becoming monolithic, a problem often encountered in C programs.

2.1.1 Simple classes

For most classes, a simple class suffices. A simple class is a class which does not inherit from other classes and from which no other classes inherit. Using mostly simple classes avoids the problems of large class hierarchies, namely the interdependence of classes which inherit from one another. The major advantage of simple classes over simple C structs is information hiding.

2.1.2 Implementing simple classes

We describe now how to implement a simple class using the string class `str.[ch]` of *GenomeTools* as an example. The interface to a class is always given in the `.h` header file (`str.h` in our example). To achieve information hiding the header file cannot contain implementation details of the class. The implementation can always be found in the corresponding `.c` file (`str.c` in our example). Therefore, we start with the following C construct to define our `GtStr` class in `str.h`:

```
typedef struct GtStr GtStr;
```

This seldomly used feature of C introduces a new data type named *GtStr* which is a synonym for the `struct GtStr` data type, which needs not to be known at this point. In the scope of the header file, the new data type `GtStr` cannot be used, since its size is unknown to the compiler at this point. Nevertheless, pointers of type `GtStr` can still be defined, because in C all pointers have the same size, regardless of their type. Using this fact, we can declare a constructor function:

```
GtStr*           gt_str_new(void);
```

which returns a new string object, and a destructor function

```
void           gt_str_delete(GtStr*);
```

which destroys a given string object. This gives us the basic structure of the string class header file: A new data type (which represents the class and its objects), a constructor function, and a destructor function.

```
#ifndef STR_H
#define STR_H

/* the string class, string objects are strings which grow on demand */
typedef struct GtStr GtStr;

GtStr*         gt_str_new(void);
void           gt_str_delete(GtStr*);

#endif
```

Now we look at the implementation side of the story, which can be found in the `str.c` file. At first, we include the `str.h` header file to make sure that the newly defined data type is known:

```
#include "str.h"
```

Then we define `struct GtStr` which contains the actual data of a string object (the *member variables* in object orientation lingo).

```
struct GtStr {
  char *cstr;           /* the actual string (always ’\0’ terminated) */
  unsigned long length; /* currently used length (without trailing ’\0’) */
  size_t allocated;     /* currently allocated memory */
};
```

Finally, we code the constructor

```
GtStr* gt_str_new(void)
{
  GtStr *s = gt_malloc(sizeof (GtStr));    /* create new string object */
  s->cstr = gt_calloc(1, sizeof (char));   /* init the string with ’\0’ */
  s->length = 0;                           /* set the initial length */
  s->allocated = 1;                        /* set the initially
                                              allocated space */
  return s;                                /* return the new string object */
}
```

and the destructor

```
void gt_str_delete(GtStr *s)
{
  if (!s) return;           /* return without action if ’s’ is NULL */
  gt_free(s->cstr);         /* free the stored the C string */
  gt_free(s);               /* free the actual string object */
}
```

Our string class implementation so far looks like this

```
#include "core/ma.h"
#include "core/str.h"

struct GtStr {
  char *cstr;           /* the actual string (always ’\0’ terminated) */
  unsigned long length; /* currently used length (without trailing ’\0’) */
  size_t allocated;     /* currently allocated memory */
};

GtStr* gt_str_new(void)
{
  GtStr *s = gt_malloc(sizeof (GtStr));    /* create new string object */
  s->cstr = gt_calloc(1, sizeof (char));   /* init the string with ’\0’ */
  s->length = 0;                           /* set the initial length */
  s->allocated = 1;                        /* set the initially
                                              allocated space */
  return s;                                /* return the new string object */
}

void gt_str_delete(GtStr *s)
{
  if (!s) return;           /* return without action if ’s’ is NULL */
  gt_free(s->cstr);         /* free the stored the C string */
  gt_free(s);               /* free the actual string object */
}
```

Since this string objects are pretty much useless so far, we define a couple more (object) methods in the header file `str.h` and the respective implementations in `str.c`.

Because C does not allow the traditional `object.methodname()` syntax often used in object-oriented programming, we use the convention to pass the object always as the first argument to the function (`methodname(object, ...)`).
To make it clear that a function is a method of a particular class *classname*, we prefix the method name with `gt_<classname>_`. That is, we get `gt_<classname>_methodname(object, ...)` as the generic form of method names in C. The constructor is always called `gt_<classname>_new()` and the destructor `gt_<classname>_delete()`. See `str.c` for examples.

2.1.3 Class scaffold code generation

The boilerplate code needed to create the structure of a new class (header and C source) can be generated automatically to avoid typical copy-and-paste errors. In the `scripts/` subdirectory of the *GenomeTools* directory tree, there is a helper script to create header files and a C source file with an implementation scaffold for a given class name. Run `scripts/codegen help` to get more information about its usage.

2.2 Interfaces

Interfaces allow several classes with possibly different implementations to share a common set of methods that can be called independently of the actual implementing class. Each implementing class must adhere to the interface method signature (that is, the return type and the number and types of parameters) but is otherwise free to implement the method as liked.
In addition to the common interface functions, a class can also have its own specific functions. To call an interface function, the object can simply be cast to the interface type, and to call an implementation-specific function, we cast it to the implementing type. The following section describes the technique used to implement interfaces in C such that objects can be cast from the interface type to the specific type without problems.
Let's imagine we want an interface called `GtExample` which has a method `gt_example_run(GtExample*)`. This corresponds to the following header file `example.h` (source files are also available in the *GenomeTools* source distribution):

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

typedef struct GtExample GtExample;

int  gt_example_run(GtExample*);
void gt_example_delete(GtExample*);

#endif
```

Note that there is no `gt_example_new()` constructor function, as the constructors will be specific to the implementing classes. Otherwise, this header is not much different to the header files for a simple class. To make the methods implementable by more than one class, we need a *class object* describing the interface-to-implementation mappings, that is, the specific functions to be called in the implementing class. This class definition is given in a `example_rep.h` header file, where "rep" stands for "representation":

```
#ifndef EXAMPLE_REP_H
#define EXAMPLE_REP_H

```

```
#include <string.h>
#include "core/example.h"

typedef struct GtExampleClass GtExampleClass;

struct GtExampleClass {
  size_t size;
  int  (*run)(GtExample*);
  void (*delete)(GtExample*);
};

struct GtExample {
  const GtExampleClass *c_class;
};

GtExample* gt_example_create(const GtExampleClass*);
void*      gt_example_cast(const GtExampleClass*, GtExample*);

#endif
```

The `GtExampleClass` stores a function pointer to the specific function implementing the `gt_example_run()` interface method. We also define a `delete` function which is called when the implementing class needs to do additional cleanup when an object of it is deleted. Given a `GtExampleClass` filled with appropriate function pointers which match the signatures, the `gt_example_create()` function then creates an object which can be cast to both the interface type `GtExample*` as well as the implementing type. To accomplish this, the size of the implementing class is needed. The reason behind this will be explained below.
Note that this header file is meant to be private, that is, it should only be included by code files which need to know about the interface-to-implementation mappings. It is then straightforward to write the `example.c` which both

- returns an object of the interface type, allocating memory to hold both a pointer to a `GtExampleClass` object (needed for calling methods in the interface context), and
- implements the interface methods by wrapping the implementation-specific function pointers given in the `GtExampleClass` object:

```
#include "example_rep.h"       /* we need access to the class struct */
#include "core/ma.h"           /* we need to allocate memory */

GtExample* gt_example_create(const GtExampleClass *ec)
{
  GtExample *e = gt_calloc(1, ec->size);   /* allocate memory */
  e->c_class = ec;                         /* assign interface */
  return e;
}

int gt_example_run(GtExample *e)
{
  gt_assert(e && e->c_class && e->c_class->run);
  return e->c_class->run(e);               /* call implementation-specific
                                              function */
}

void gt_example_delete(GtExample *e)
```

```
{
  if (!e) return;
  gt_assert(e && e->c_class);
  if (e->c_class->delete != NULL) {
    e->c_class->delete(e);                /* delete implementation-specific
                                             members */
  }
  gt_free(e);                             /* delete interface */
}
```

Now, let us have a look at how the implementing classes are written. Let's imagine we want class `GtExampleA` to implement the `GtExample` interface. Of course, we need a class header file `example_a.h` containing a constructor and destructor, just as described in section 2.1.2:

```
#ifndef EXAMPLE_A_H
#define EXAMPLE_A_H

typedef struct GtExampleA GtExampleA;

GtExample*  gt_example_a_new();

#endif
```

An implementation of the `GtExampleA` class in the `example_a.c` source file then contains the code for the specific methods and their assignment to the interface mapping. First, we need to include the headers to be able to register our implementation-specific functions in the mapping struct:

```
#include "example_a.h"
#include "example_rep.h"
```

Then, we define our `GtExampleA` class as usual, but leave enough space for an instance of the interface class at the beginning of our definition:

```
struct GtExampleA {
  GtExample parent_instance;
  unsigned long my_property;
};
```

By placing an instance of the interface at the beginning of our implementation, we allow the same pointer (to the beginning of the data structure) to be cast to

1. a pointer to a `GtExample` interface implementation, so it can be used safely with the `gt_example_*()` interface methods, restricting access to the interface members only, and

2. a pointer to the `GtExampleA` data structure, which can safely be used with the `gt_example_a_*()` methods, ignoring the interface part and allowing access to the implementation member variables only.

Figure 1 illustrates this concept.
In the rest of `example_a.c`, we then code our implementation of the `run` interface method:

```
static int gt_example_a_run(GtExample *e) /* hidden from outside  */
{
  GtExampleA *ea = (GtExampleA*) e;       /* downcast to specific type */
  printf("%lu", ea->my_property);         /* run functionality */
  return 0;
}
```

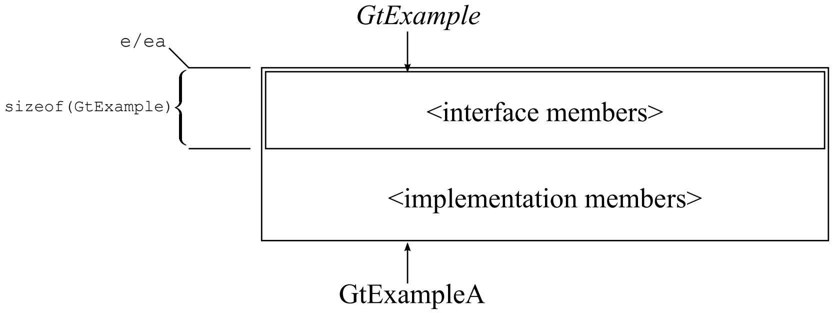

Figure 1: Memory layout used in the `GtExampleA` object starting at the memory location `e` implementing the `GtExample` interface.

Note that we cast our generic `GtExample*` pointer into a more specific `GtExampleA*` pointer. We can do this because we can now be sure that this function has been called on an object of the `GtExampleA` class. We can be sure because we have registered this method as an implementation of the `run` interface method by assigning it to the function pointer variable in the `GtExampleClass` structure:

```
/* map static local method to interface */
const GtExampleClass* gt_example_a_class(void)
{
  static const GtExampleClass ec = { sizeof (GtExampleA),
                                     gt_example_a_run,
                                     NULL };
  return &ec;
}
```

Note that we assign NULL to the `delete` function slot, because we do not allocate any memory inside the implementing class we need to free later. Have a look at the `example_b.*` files in the *GenomeTools* source distribution for an alternative implementation which allocates additional memory.
We can then use the `GtExampleClass` returned by this function to write the `GtExampleA` constructor, which uses `gt_example_create()` to allocate the needed space, initializes the private members and returns the object:

```
GtExample* gt_example_a_new(void)
{
  GtExample *e = gt_example_create(gt_example_a_class());
  GtExampleA *ea = (GtExampleA*) e;      /* downcast to specific type */
  ea->my_property = 3;                   /* access private implementation
                                            member */
  return e;
}
```

Now consider another implementation, `GtExampleB` which also implements this interface by creating a `GtExampleClass` with different implementation- specific function pointers (see the `example_b.*` files in the distribution).
Combining these implementation with the interface headers now allows us to do the following:

```
#include "example.h"       /* include the interface header */
#include "example_a.h"     /* include the implementation header */
#include "example_b.h"     /* include another implementation header */

int main(int argc, char *argv[])
{
  GtExample *my_e = gt_example_a_new();        /* create GtExampleA object, but
                                                  with interface type */
  gt_example_run(my_e);                        /* call an interface method */
  gt_example_delete(my_e);

  GtExample *my_e = gt_example_b_new();        /* create GtExampleB object, but
                                                  with interface type */
  gt_example_run(my_e);                        /* call an interface method */
  gt_example_delete(my_e);

  return 0;
}
```

That is, we can access two implementations via a common set of interface methods.

2.3 Modules

Modules bundle related functions which do not belong to a class. Examples:

- `dynalloc.h`, the low level module for dynamic allocation, e.g. used to implement arrays in `array.c` and the above-mentioned strings
- `sig.h`, bundles signal related functions (high level)
- `xansi.h`, contains wrappers for the standard ANSI C library
- `xposix.h`, contains wrappers for POSIX functions we use

When designing new code, it is not very often the case that one has to introduce new modules. Usually defining a new class is the better approach.

2.4 Unit tests

Classes and modules should contain a `gt_<classname>_unit_test` function which performs a unit test of the class/module and returns 0 in case of success and -1 in case of failure. More information about how to write unit tests can be found in section 9.1.

2.5 Tools

A *tool* is the most high-level type of component *GenomeTools* has to offer. Tools are command line interface (CLI) applications linked into the single `gt` binary. They make use of helper classes like the `GtOptionParser` to make development of command line tools easier. Having a common interface for option parsing and error reporting ensures a consistent user experience across all *GenomeTools* tools, as they behave the same way when invoked from the command line.

There are two possible code paths for defining and implementing a tool; only the newer approach will be described here. Simply put, a tool is just another object which needs to implement a special interface, providing callbacks for the *GenomeTools* runtime to call at predefined times during the tool's invocation. An example for a simple tool can be found in the `tools` subdirectory in the `gt_template.[ch]` files.

First, a tool needs to define a structure to store its arguments. That is, every command line parameter needs to be represented by a member in the struct to store its value. For example, for a tool taking a boolean and a string parameter, we would need the following:

```
typedef struct {
  bool bool_option;
  GtStr  *str_option;
} ExampleToolArguments;
```

The tool also requires an initializer function which prepares the argument structure for value assignment. For example, the `GtStr` in above example must be instantiated:

```
static void* gt_example_tool_arguments_new(void)
{
  ExampleToolArguments *arguments = gt_calloc(1, sizeof *arguments);
  arguments->str_option = gt_str_new();
  return arguments;
}
```

as well as a destructor function which deletes the argument objects, if necessary, and then frees the memory used for the argument struct:

```
static void gt_example_tool_arguments_delete(void *tool_arguments)
{
  ExampleToolArguments *arguments = tool_arguments;
  if (!arguments) return;
  gt_str_delete(arguments->str_option);
  gt_free(arguments);
}
```

The argument structure is filled by an *option parser*. An option parser is an object which gets passed an argument list, identifying parameter names and values and assigning them to the correct variables. It also handles the creation of a convenient help output by documenting the purpose of each option and its valid value range. See the interface documentation in `src/core/option.h` for a list of possible option types. A tool must contain a function returning an option parser object:

```
static GtOptionParser* gt_example_tool_option_parser_new(void *tool_arguments)
{
  ExampleToolArguments *arguments = tool_arguments;
  GtOptionParser *op;
  GtOption *option;
  gt_assert(arguments);

  /* initialize with one-liner */
  op = gt_option_parser_new("[option␣...]␣[file]",
                            "This␣is␣an␣example␣tool␣for␣demonstration␣"
                            "purposes.");

  /* -bool */
  option = gt_option_new_bool("bool",
                              "this␣is␣the␣boolean␣option",
                              &arguments->bool_option,
                              false);        /* default value */
  gt_option_parser_add_option(op, option);

  /* -string */
  option = gt_option_new_string("string",
                                "pass␣any␣string␣here",
                                arguments->str_option,
                                NULL);       /* default value */
  gt_option_parser_add_option(op, option);
  return op;
}
```

The option parser already performs initial validation of the parameters. For example, it makes sure that a numeric parameter is not given a string value, that unsigned values are always positive or that probabilities stay between 0 and 1. In an error case, tool invocation is stopped and the appropriate error message is printed to `stderr`.

For more sophisticated error checking, for example involving several parameters and their values at once, is is possible to write an argument checking function, which can set an error message in a `GtError` object (see 6.2) and return a non-zero return value if an error was found:

```
static int gt_example_tool_arguments_check(GT_UNUSED int rest_argc,
                                           void *tool_arguments,
                                           GT_UNUSED GtError *err)
{
  ExampleToolArguments *arguments = tool_arguments;
  int had_err = 0;
  gt_error_check(err);
  gt_assert(arguments);

  /* we assume that the string parameter must not be empty */
  if (gt_str_length(arguments->str_option) == 0) {
    gt_error_set(err, "parameter␣’string’␣must␣not␣be␣empty!");
    had_err = -1;
  }

  return had_err;
}
```

In most cases, however, this function is not necessary and needs not be implemented.

The most important function which must be implemented in a tools is the runner. The runner calls the code that actually performs the tool’s function and is the equivalent to the `main` function in a traditional C program. Its signature is very similar to a typical C `main` function as well, being passed the number of arguments `argc` and an array of argument strings `argv`.:

```
static int gt_example_tool_runner(int argc, const char **argv,
                                  int parsed_args,
                                  void *tool_arguments,
                                  GT_UNUSED GtError *err)
```

In addition, it receives the number of arguments (`parsed_args`) which were already parsed by the option parser, thus specifying an offset in the argument array from which the rest of the arguments begin. That is, if the parameter string was

```
-bool true -string foo bar baz
```

then `parsed_args` would be 4, as the `-bool` and `-string` options and their values have already been parsed, leaving `argv[parsed_args] = ’bar’` and `argv[parsed_args+1] = ’baz’` to be handled by the runner.

The rest of the runner function could look like this:

```
static int gt_example_tool_runner(int argc, const char **argv,
                                  int parsed_args,
                                  void *tool_arguments,
                                  GT_UNUSED GtError *err)
{
  ExampleToolArguments *arguments = tool_arguments;
  int had_err = 0;
  gt_error_check(err);
  gt_assert(arguments);

  if (arguments->bool_option)
    printf("the␣bool␣option␣was␣set\n");
  printf("the␣string␣was␣'%s',␣gt_str_get(arguments->bool_option));

␣␣return␣had_err;
}
```

Finally, the functions described above are registered in the new tool object by using `gt_tool_new()` to create a new `GtTool` instance passing pointers to all the static callback functions.

```
GtTool* gt_example_tool(void)
{
  return gt_tool_new(gt_example_tool_arguments_new,
                     gt_example_tool_arguments_delete,
                     gt_example_tool_option_parser_new,
                     gt_example_tool_arguments_check,
                     gt_example_tool_runner);
}
```

Let's assume that we have saved the implementation above in `tools/gt_example_tool.c`. We then make the `gt_example_tool()` function public by adding a `tools/gt_example_tool.c` header:

```
#ifndef GT_EXAMPLE_TOOL_H
#define GT_EXAMPLE_TOOL_H

#include "core/tool.h"

/* the example tool */
GtTool* gt_example_tool(void);

#endif
```

This function can then be added to the *GenomeTools* toolbox by adding the following lines to `gtt.c`:

```
...
#include "tools/gt_example_tool.h"
...
GtToolbox* gtt_tools(void)
{
  ...
  gt_toolbox_add_tool(tools, "example", gt_example_tool());
  ...
}
```

After compilation, we can then run our tool by calling

```
$ gt example -bool true -string foo bar baz
```

3 Directory structure

All of these directories are given as subdirectories of the root directory of the *GenomeTools* source distribution.

`bin/` This subdirectory contains the *GenomeTools* binary executable `gt` as dynamic and static variants as well as the example executables built from `src/examples`. This directory is only populated after a make run. Running `make cleanup` will remove its contents.

`doc/` This subdirectory contains documentation such as this developer's guide, license information, format specifications, and the user manuals for the software tools included with the *GenomeTools*.

`gtdata/` This subdirectory contains data needed for the *GenomeTools* to run which are not compiled into the *GenomeTools* binary itself, such as

- texts for the tool on-line help (in `gtdata/doc`),
- Lua code for documentation generation (in `gtdata/modules`),
- ontology definition files (in `gtdata/obo_files`),
- *AnnotationSketch* default (e.g. a default style file, in `gtdata/sketch`), and
- alphabet definition files for character mappings (in `gtdata/trans`).

`gtpython/` This subdirectory contains the Python bindings to selected parts of the *GenomeTools* library, as well as the Python test suite. See the `README` file in this directory for installation instructions.

`gtruby/` This subdirectory contains the Ruby bindings to selected parts of the *GenomeTools* library. See the `README` file in this directory for installation instructions.

`gtscripts/` This subdirectory contains a number of Lua scripts written using the *GenomeTools* Lua bindings; most prominently the `gtdoc.lua` script to generate the documentation. These scripts can be run using the `gt` executable, which is a Lua interpreter as well, by giving the script name instead of a tool name.

`lib/` This subdirectory contains the *GenomeTools* static and dynamic libraries when built.

`obj/` This subdirectory contains object files as they are created during *GenomeTools* compilation.

`scripts/` This subdirectory contains useful scripts for *GenomeTools* developers.

`src/` This subdirectory contains the main *GenomeTools* source tree. In particular, there is a number of subdirectories:

- the `src/annotationsketch` subdir contains *AnnotationSketch* code for genome annotation drawing,
- the `src/core` subdir contains general code, i.e. basic data structures, memory management, file access, encoded sequences, sequence parsers, tool runtime, option parser, multithreading, etc.,
- the `src/examples` subdir contains simple example applications built on *GenomeTools* (streams, or a GUI app),
- the `src/extended` subdir contain code for annotation handling and parsing, stream processing, alignment, chaining, etc.,
- the `src/external` subdir contains third-party source code which is distributed with the *GenomeTools* source and built alongside the *GenomeTools*,

- the `src/gth` subdir with *GenomeThreader* code,
- the `src/gtlua` subdir with Lua bindings for some of the *GenomeTools* classes and modules,
- the `src/ltr` subdir with LTR retrotransposon prediction and annotation code,
- the `src/match` subdir with code for index structure construction and access, short read mapping, matching algorithms etc.,
- the `src/mgth` subdir contains *MetaGenomeThreader* code,
- the `src/patches` subdirectory with platform-specific patches, and
- the `tools` subdir with code for all the tools included with *GenomeTools*.

`testdata/` This subdirectory contains test data used in the testsuite. Please refrain from storing large files (> 1MB) in this directory, but use the `gttestdata` repo instead (see 9.2.2). Special subdirectories:

- The `testdata/gtscripts` subdir contains test scripts used in the Lua test cases,
- the `testdata/gtruby` subdir contains test scripts used in the Ruby test cases, and
- the `testdata/gtpython` subdir contains test scripts used in the Python test cases.

`testsuite/` This subdirectory contains the test suite definitions as Ruby files as well as the test engine and temporary data created using test runs. After starting a test suite run, the `testsuite/stest_testsuite` subdirectory then contains a directory named `test<n>` for each test, where `<n>` is the test number. See 9.2.1 for more details.

`www/` This subdirectory contains the content of the *GenomeTools* website.

4 Public APIs

In *GenomeTools*, we distinguish between *public* and *non-public* application programming interfaces (APIs). The API describes the classes and modules belonging to the *GenomeTools* and their methods and functions, in particular their signatures; that is, their name, return value, and number and types of their parameters.

The public API is a subset of the *GenomeTools* library which is intended to be used by developers which do not belong to the *GenomeTools* core development team, and is fairly high-level at this point. To ensure compatibility with future versions of the *GenomeTools* library, the public API is supposed to be subject to as little change as possible. That is, interface changes should be made very sparsely, and interface design should 'look forward' to make such changes unneccessary. For example, interface functions which could fail in theory should receive error handling facilities in their signature (such as a return code and a `GtError` object), even if their current (and maybe only) implementation cannot fail. This leaves room for implementations that *may* fail without having to break the API when the new implementation finds its way into the *GenomeTools*.

All public API functions for a given class must be declared in a prototype header file named `<class>_api.h`. This header file must only include other public API headers. All of the public API header files are packaged to be distributed with the *GenomeTools* tarball and are installed into the given include path. That is, the functions defined in them are later accessible by including `genometools.h` only.

It is important to note that all functions in the public header files must be properly documented (see section 5.7).

5 Coding style

5.1 General rules

- No line in the source code must be longer than 80 characters. This allows proper formatting of the code.

- There must be not more than one consecutive empty line in the source code.

- Trailing spaces are disallowed.

- There must not be a comma at the beginning of a line.

- Unless it is at the end of a line, a comma should be followed by a space. Example:

  ```
  cmpfunc(gt_array_get(a, idx), gt_array_get(b, idx));
  ```

  instead of

  ```
  cmpfunc(gt_array_get(a,idx),gt_array_get(b,idx));
  ```

- The symbols ‘`=`’, ‘`==`’ und ‘`!=`’ should be enclosed by spaces. That is, write

  ```
  i = 0;
  ```

  instead of

  ```
  i=0;
  ```

- There must be a space between the keywords `for`, `if`, `sizeof`, `switch`, `while` and `do` and the following parenthesis.

- The opening braces (`{`) after the keywords `if`, `else`, `for`, `do`, and `while` should be on the same line as the keyword.

- The curly braces following an `if` or `else` expression should be omitted if the expression and the (single) following statement both fit on a single line.

- The keyword `else` should be placed on a separate line.

- Semantic blocks (statements inside loops, function definitions, etc.) must be indented by exactly two spaces w.r.t. the enclosing block. This explicitly means *no tabs*, configure your editor!

  Here is an example:

  ```
  bool
  gt_array_equal(const GtArray *a, const GtArray *b, GtCompare cmpfunc)
  {
    unsigned long idx, size_a, size_b;
    int cmp;
    gt_assert(gt_array_elem_size(a) == gt_array_elem_size(b));
    size_a = gt_array_size(a);
    size_b = gt_array_size(b);
    if (size_a < size_b)
      return false;
    if (size_a > size_b)
      return false;
  ```

```
  for (idx = 0; idx < size_a; idx++) {
    cmp = cmpfunc(gt_array_get(a, idx), gt_array_get(b, idx));
    if (cmp != 0)
      return false;
  }
  return true;
}
```

- Use the `scripts/src_check` and `scripts/src_clean` scripts regularly to check your source code for style violations.
- Use the `scripts/pre-commit` git hook to automatically run a `src_check` before each commit. The commit will be canceled if errors are found.
  To enable the git hook, copy the file `scripts/pre-commit` into the `.git/hooks` subdirectory of your *GenomeTools* repository.
- Static variables inside functions are not allowed. An exception are class structs, which must be static.
- All functions except those which should be callable publicly should be declared as `static`. All non-static functions must be documented in a header file. Think twice before making a function public. Its interface should be clean enough to be understood by someone who does not know implementation details!
- If a *GenomeTools* module or class exists for your particular need, use it instead of using more low-level means (e.g. try to use `GtFile` and friends for file access instead of `fopen()`/`fclose()`/... directly). Consult the documentation and the header files!

5.2 Global variables

- Generally, global variables are not allowed.
- There are exceptions in very rare cases, in which must be made sure that the content in question is initialized, synchronized for multithreaded use and properly cleaned up. Do not add global variables without talking to one of the core developers!

5.3 Types

- Use unsigned types whereeever possible. We need to process large amount of data and we may need every bit to process it.
- Use `unsigned long` for sequence positions/lengths/offsets/.... The `unsigned long` type equals the word size on all common systems (i.e., it is 32-bit wide on 32-bit systems and 64-bit wide on 64-bit systems) which makes it ideal for most use cases.
- Use `GtStr` and `GtArray` instead of manipulating byte arrays directly if possible, especially when returning strings or item collections from a function.

5.4 Naming rules

- Class names must begin with `Gt`, e.g. `GtArray`, `GtNodeStream`.
- Class names may use camel case[1] if multiple words are required, e.g. `GtNodeStream`.
- Source and header files for a class must start with the class name in lowercase, without the `Gt` prefix. If camel case is used in the class name, use underscores in the respective file name, e.g. `GtNodeStream` → `node_stream.[ch]`.
- Variable names and function names must be all lower-case.
- Variable names and function names should use underscores to separate words (e.g. use the function name `get_first_five_chars` instead of `getfirstfivechars`).
- The names of public (i.e. non-static) functions must be prefixed by the string '`gt_`' to avoid namespace clashes when linking the *GenomeTools* library with third-party code.
- Class or module names must follow the '`gt_`' part in the function name.
- In method signatures, the object on which the method is called must always be the first argument of the method. That is, method `method` in class `GtClass` must be defined as:

  ```
  <type> gt_class_method(GtClass*, <params>);
  ```

5.5 Copyright lines

- Every header and C source file must begin with a comment containing author and license information:

  ```
  /*
  Copyright (c) 2007-2010 Gordon Gremme <gremme@zbh.uni-hamburg.de>
  Copyright (c) 2007-2008 Center for Bioinformatics, University of Hamburg

  Permission to use, copy, modify, and distribute this software for any
  purpose with or without fee is hereby granted, provided that the above
  copyright notice and this permission notice appear in all copies.

  THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
  WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
  MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
  ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
  WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
  ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
  OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
  */
  ```

- Each developer with substantial contributions (for example, by implementing new features, refactoring, or fixing bugs which require large rewrites) should give his/her name and email address, updating the given year ranges in the process.
- The ”Center for Bioinformatics” copyright line is only required for developers employed by it, because only in this case the University gains any copyright.

[1] `http://en.wikipedia.org/wiki/CamelCase`

5.6 Header files

- Use conditional inclusion to avoid including the same header file multiple times. Between the copyright header and the beginning of the declarations, put the following:

  ```
  #ifndef FILENAME_H
  #define FILENAME_H
  ```

  and put an

  ```
  #endif
  ```

  at the end of the header file. This causes the preprocessor to include the section between the `#ifndef` and `#endif` only once.

- The identifier to be `#define`d must be the filename of the header file in uppercase with all non-alphanumeric characters replaced by underscores.

5.7 Comments and documentation

- Comments are written in plain C style (`/* ... */`). C++-style comments (`// ...`) are disallowed.
- The public API files (header files ending in `*_api.h`) are examined for automatic generation of API documentation. The following annotations are supported:
  - Each type (e.g. class) should be directly preceded by a comment describing the purpose of the class. Example:

    ```
    /* <GtArray*> objects are generic arrays for elements of a certain
       size which grow on demand. */
    typedef struct GtArray GtArray;
    ```

  - Each function (method) should be directly preceded by a comment describing the purpose of the function, its preconditions, return value, and parameters. Example:

    ```
    /* Add element <elem> to <array>. The size of <elem> must equal the
       given element size when the <array> was created and is determined
       automatically with the <sizeof> operator. */
    #define        gt_array_add(array, elem) \
                   gt_array_add_elem(array, &(elem), sizeof (elem))
    /* Add element <elem> with size <size_of_elem> to <array>.
       <size_of_elem> must equal the given element size when the <array>
       was created. Usually, this method is not used directly and the
       macro <gt_array_add()> is used instead. */
    void           gt_array_add_elem(GtArray *array, void *elem,
                                     size_t size_of_elem);
    ```

  - Code keywords (parameters, class names, references to other functions) can be marked by putting them between angled brackets (<. . .>). Also, keywords can be marked as strong (bold) by putting them between three underscores (___. . . ___) or emphasized (italic) by using two underscores (__. . . __).
- The comment line should briefly describe
  - what the method does (e.g. "Calculates and returns X. . . ", "Delivers the next element in order. . . ", "Adds element X. . . "),

  - what *all* parameters are supposed to be (ideally given in the context of what the method does),
  - what the return value is,
  - potential side-effects, and
  - if the function receives or returns a pointer, whether ownership is taken or retained for the accepted or returned memory (i.e. “takes ownership of parameter X” or “X must be deleted by the caller”).

- Do not group multiple functions beneath one comment line, even if they are largely similar and differ only in minor details. If required, repeat the comment above each function to ensure that both get an entry in the generated documentation.

- Stars (`*`) are *not* to be continued on every line of the comment, as many editors like to do by default. Doing so anyway will lead to interspersed star symbols in the generated output.

- Please refrain from using other markup formats such as Doxygen or Javadoc style. It may look alright in the plain text, but will most certainly come out weird in the generated LaTeX or HTML documents.

- Use the `docs` target in the *GenomeTools* Makefile to build the API documentation, which is then found in `www/genometools.org/htdocs` subdirectory (libgenometools.html).

5.8 Function pointers

- Function pointer declarations used in public headers (e.g. to be used as method arguments) should always be `typedef`’ed to an identifier prefixed with `Gt`. An example:

  ```
  typedef int  (*GtCompareWithData)(const void*, const void*, void *data);
  ```

  The function pointer type can then be used in a sort function this way:

  ```
  void gt_qsort_r(void *a, size_t n, size_t es, void *data,
                  GtCompareWithData cmp);
  ```

  instead of:

  ```
  void gt_qsort_r(void *a, size_t n, size_t es, void *data,
                  int (*cmp)(const void*, const void*, void *data));
  ```

  which makes the headers much harder to read.

- If callback functions need additional data to work, provide an additional `void*` pointer to pass external data to the callback function (as, for example, done with the `data` parameter in the function `gt_qsort_r()` above). Do not use global variables for that purpose!

5.9 `#defines`

- Identifiers introduced by a `#define` statement should be completely written in upper case letters. This also holds for the arguments of macros.

- Identifiers introduced by public `#define`s should be prefixed with `GT_`.

- In public headers, `#define` statements should only set constants. Macros defining code parts should only be used – very sparingly – locally within a class or module implementation.

5.10 Information hiding

In *GenomeTools*, we try to adhere to object-oriented design guidelines. An integral part of these is the enforcement of *information hiding* or *encapsulation*, that is, making access to an object's internal state only possible through accessor functions.

From this, it follows directly that the use of 'open' structs, that is, C structures defined in public headers, is strongly discouraged! Structures (and structure member accesses) should only be implemented within a C file. This applies to both structures used to implement classes as well as structures used as auxiliary data structures.

A notable exception are structs implementing classes which only act as containers, i.e. in which the only methods manipulating the members would be setter and getter methods (see `core/range_api.h`). However, even those structures should provide accessor, constructor and destructor methods so they can safely be created, accessed and deleted from scripting language bindings lacking support for proper C structure access.

6 Error handling

We distinguish between *programming errors* and *run-time errors*. Programming errors occur when the developer uses resources in an inherently erroneous way, e.g. passing a null pointer where a valid pointer is expected, passing incorrect length values to string comparison functions, or accessing uninitialised memory. This often happens because of incomplete documentation, or by running into "corner cases" which were overlooked. If undetected, they can lead to nasty bugs that are hard to track down. Other special cases are expected to come up sooner or later. For example, a user may specify a file name for input or output which is not readable or writable, or which does not exist. Typically, it is not intended to terminate program execution at that point, but to react in a graceful way (e.g. by creating a new file, reporting a proper error message, or asking for a different file name). These are run-time errors.

6.1 Programming errors

- Use `gt_assert()` to check invariants in your program, that is, any conditions that you hold to be true for the following code to work properly. For example, in a function getting a pointer as a parameter which is to be dereferenced later, you should assert that the pointer is not `NULL` prior to dereferencing. Similarly, in a function that is supposed to never return a negative number, one should use `gt_assert()` to check this condition directly before the `return` statement.

- If the expression given to `gt_assert()` as a parameter evaluates to `false` (that is, 0, `false`, or `NULL`), the program will abort.

- Assertions can be disabled at compile time by passing the `assert=no` switch to the `make` call. If variables or parameters are only used in an assertion, disabling assertions may trigger this error message:

  ```
  error: unused parameter 'x'
  ```

  or

  ```
  error: unused variable 'y'
  ```

Include core/unused_api.h and prefix the declaration of the offending identifier with GT_UNUSED to inform the compiler that this variable is intentionally unused.

6.2 Run-time errors

- Functions that are allowed to fail at run-time must return a negative error code or NULL. The code for successful execution should be 0 or a pointer different from NULL. Positive return values may be used as results. Such functions should also receive a GtError object as their last parameter, in order to store and propagate error messages. If a function may return an error code or NULL, *always* check for this and handle the case accordingly.

- Use the GtError class (see core/error_api.h) for storing error messages and error status:

```
char* get_first_five_chars(const char *str, GtError *err)
{
  char *ret;
  if (strlen(str) < 5) {
    gt_error_set(err, "string␣'%s'␣is␣shorter␣than␣5␣characters", str);
    return NULL;
  }
  ret = gt_calloc(6, sizeof (char));
  strncpy(ret, str, 5);
  return ret;
}
```

- Use a variable called had_err which is initialized to 0 and then assigned an error status such that an error is set when had_err$\neq$ 0:

```
int had_err = 0;
const char *prefix;
GtError *err = gt_error_new();
if ((prefix = get_first_five_chars("foo", err))) {
  /* go on with next step */
}
else
  had_err = -1;
if (!had_err) {
  ...
}
```

  -1 should be used to store an error in the had_err variable. It is very idiomatic to write if (had_err)... or if (!had_err)....

- Catch run-time errors and create error messages as close to their source as possible.

- The error object should always be the last parameter and should be named err.

7 Memory management

7.1 Allocation/deallocation

- Space allocation is only allowed using the gt_malloc(), gt_calloc(), gt_realloc() functions in core/ma_api.h. Use gt_free() to deallocate memory. These methods are analog to malloc(3), calloc(3) and realloc(3) from the C standard library, except that they never return NULL upon failure.

- The gt_ma_get_space_peak(), gt_ma_show_space_peak() and gt_ma_check_space_leak() functions in core/ma_api.h can be used to evaluate memory usage and check for memory leaks.

7.2 Reference counting

Sometimes it is desired to have an object referenced by more than one other object, avoiding to gt_free() the object's memory until the last reference to the object has been dropped. In *GenomeTools*, *reference counting* is used to implement this behaviour. That is, each object keeps the number of objects still keeping a reference on it in a local member variable. This is done by each referencing object calling the object's ref() method to announce that they now keep a reference, thus increasing the reference count. When the object reference is no longer needed, the usual delete() method is used. The delete() method checks and decreases the reference count and defers free'ing the object's memory until the object is not referenced by any other object any more. This makes reference counting a simple form of garbage collection.
To add reference counting to a class, perform the following steps:

- Add an unsigned int counter variable called reference_count to the private member variables of the class; this variable must be initialized to 0 in the constructor.

- Consider a class *GtFoo* to which we want to add reference counting capabilities. Then add a method gt_<classname>_ref(), in this case gt_foo_ref() to the interface of the class:

```
GtFoo* gt_foo_ref(GtFoo *f)
{
  gt_assert(f);
  f->reference_count++;
  return f;
}
```

- In the destructor, check the reference count and only free the memory when necessary:

```
void gt_foo_delete(GtFoo *f)
{
  if (!f) return;
  if (f->reference_count) {
    f->reference_count--;
    return;
  }
  gt_free(f);
}
```

With reference counted classes, always use the ref() method when storing a reference to an object. That is, instead of writing

```
void gt_bar_set_foo(GtBar *b, GtFoo *f)
{
  b->value = f;
}
```

use

```
void gt_bar_set_foo(GtBar *b, GtFoo *f)
{
  b->value = gt_foo_ref(f);
}
```

Always remember to call gt_foo_delete() when the reference is no longer needed! For example, in the assignment above, a good place to do this is the destructor of the GtBar class.

7.3 Library initialization/finalization

Within the *GenomeTools* code, there are a number of global or static data which must be properly initialized before using any *GenomeTools* functionality, and space for which must be properly freed when done using the *GenomeTools*. This is usually done by the runtime by calling initializers when a tool is run using the gt binary.

Now consider that *GenomeTools* can also be used as a library, called libgenometools. That means, it is possible to link an external code with the static or shared object file and call functions from there, without going through the tool runtime. It is now crucial that the necessary initializations have taken place before using functions, and that the required cleanup is done at the end.

There are two functions in the core/init.[ch] module in *GenomeTools* used to accomplish this:

- gt_lib_init(), which initializes all static data and should be called before any other *GenomeTools* function, and
- gt_lib_clean(), which frees all static data. It returns 0 if no memory map, file pointer, or memory has been leaked and a value other than 0 otherwise.

It is also possible to make the cleanup happen automatically when the program using the library exits. This is done by calling gt_lib_reg_atexit_func() which registers an exit handler with the OS which will call gt_lib_clean() automatically.

8 Threads

The *GenomeTools* contain functions allowing developers to write their programs in a multi-threaded way, by wrapping the POSIX threads library libpthread. See core/thread.h for more information.

Any function with this signature:

```
void* (*GtThreadFunc)(void *data);
```

can be enabled to be run concurrently by simply calling

```
void *mythread(void *data)
{
 ...
}
gt_multithread(mythread, NULL, err);
```

Some more useful information:

- For synchronization during parallel execution of multiple threads, *GenomeTools* provides classes for mutexes and read-write-locks. See core/thread.h for a description of the interface of the GtMutex and GtRWLock classes.

- If code must be conditionally compiled depending on thread support, use #ifdef and friends with the GT_THREADS_ENABLED flag, which is set by the compiler via a -D option when threading support is enabled.
- Threading support is enabled at compile time by passing threads=yes to the make call. If threading support is not enabled, all gt_multithread() functions will run the thread function sequentially.
- If threading support is enabled, the number of concurrent jobs can be given using the -j parameter to the gt binary. That is, to have all multithreaded parts in the tool to be run use three threads at once, call the tool with

  ```
  $ gt -j 3 <toolname> ...
  ```

9 Testing and Debugging

9.1 Testing on the code level – the unit tests

Unit test check whether classes and their methods behave correctly when used in a correct manner. The corresponding functions must be defined in the class implementation file (so they get access to the private member variables of the tested class) and adhere to the following interface:

```
int  (*UnitTestFunc)(GtError *err);
```

They must return 0 if the test was successful and -1 if the test has failed. The gt_ensure helper macro makes writing unit tests easier. To use it, #include the file core/ensure.h. Then write your unit test:

```
int gt_class_unit_test(GtError *err) /* must be called 'err'! */
{
  int had_err = 0;
  gt_error_check(err);               /* will abort if error was already set */

  gt_ensure(had_err, 1 + 1 == 2);    /* will succeed */
  gt_ensure(had_err, 1 + 1 == 3);    /* will fail */

  return had_err;
}
```

Similarly to gt_assert(), if the expression given to gt_ensure() as a second parameter evaluates to false, the test will fail with an error message, giving the location at which the first condition failed. Because gt_ensure() is implemented as a macro relying on certain naming conventions, it is mandatory that the GtError object passed to the test function is called err.

The unit tests are added to the test suite in the function gtt_unit_tests() in gtt.c and loaded into the *GenomeTools* runtime in the function gtr_register_components() in gtr.c. That is, if your unit test function is gt_class_unit_test(), then you should #include the header class.h (which contains the function prototype) in gtt.c and add the following line in gtt_unit_tests():

```
gt_hashmap_add(unit_tests, "example␣class", gt_class_unit_test);
```

The tests registered in this hash table can be executed on the command line with:

```
$ gt -test
```

It is also possible to run a single test from the test suite by using the -only option:

```
$ gt -test -only 'example class'
```

9.2 Testing on the tool level – the test suite

While the unit tests check the correctness of the classes and modules on the code level, the Ruby-based test suite is used to run tests on the tools themselves. That means that they run tools with example data or invalid parameters and check whether they behave correctly by looking at error levels, error messages, and comparing output with reference data.
Test data and reference data are stored in the `testdata/` directory of the *GenomeTools* source tree. Note that this directory is for smaller test data only. Large files, such as whole chromosome annotations go into another repository (see 9.2.2).

9.2.1 Test definitions

Tests are defined in the `gt_<toolname>_include.rb` files in the `testsuite/` directory. They contain test definitions written in a Ruby-based domain specific language. Here is an example of a simple test:

```
Name "gt␣cds␣test␣(description␣range)"
Keywords "gt_cds␣usedesc"
Test do
  run_test "#{$bin}gt␣cds␣-usedesc␣-seqfile␣" +
           "#{$testdata}gt_cds_test_descrange.fas␣" +
           "#{$testdata}gt_cds_test_descrange.in"
  run "diff␣#{$last_stdout}␣#{$testdata}/gt_cds_test_descrange.out"
end
```

This test runs the `gt cds` command with example data and compares its output on stdout with a reference file.
Every test case must have a `name` and can have a set of *keywords* associated with it, allowing for selective running of a subset of tests from all testsuites. Keywords are separated by spaces. The actual test code is given in the `Test` environment. Within this environment, one may use the following constructs to define test conditions:

- `run_test(runstring, options)`, where
  - `runstring` is the tool commandline to run, and
  - `options` are a hash specifying test constraints. The key `:retval` specifies the expected error code for this run (0 is the default). The option `:maxtime` specifies the maximal time in seconds that the started program may run before it is killed, resulting in a failed test (60 is the default). This allows one to detect infinite loops without stopping the testing progress.

  This command runs the command specified in `runstring`, and causes the test to fail if the returned error code does not equal the expected one.

- `grep(file, pattern)` which searches for `pattern` in the file `file`, failing the test if there is no match. The pattern can be given as a regular expression.

- `run_ruby(rubyscript, options)` and `run_python(pythonscript, options)` can be used to run tests on external Ruby and Python scripts.

- Any Ruby code, such as custom functions, can be run inside the `Test` environment. To fail a test case manually, use the `failtest(msg)` command, where `msg` is the error message to fail with.

Inside test suite definitions, some useful paths are predefined to be conveniently used in test runs (like `$bin` and `$testdata` in the example above):

- `$testdata`, the path to the `testdata/` directory,
- `$gttestdata`, the path to the location of the `gttestdata` repository (see 9.2.2),
- `$bin`, the path to the *GenomeTools* `bin/` directory,
- `$cur`, the path to the current test case directory (each test is run in a separate directory),
- `$transdir`, the path to the `gtdata/trans` directory,
- `$obodir`, the path to the `gtdata/obo_files` directory,
- `$gtruby`, the path to the `gtruby/` directory,
- `$gtpython`, the path to the `gtpython/` directory,

It is also possible to get the standard output and standard error contents of the last command run by referring to the files specified by `$last_stdout` and `$last_stderr`.
Furthermore, each test commandline (let's say the *i*-th one in the test) creates a set of `run_`*i* (contains the actual command which was run), `stdout_`*i* (contains the standard output) an `stderr_`*i* (contains the standard error output) files in the test directory (which is `testsuite/stest_testsuite/test`*n*`/`, where *n* is the test number (printed in front of each test name).

9.2.2 The `gttestdata` repository

Large test or reference data must not be placed into the *GenomeTools* `testdata/` directory because they would increase the size of the *GenomeTools* distribution too much. For such data there is a separate repository, which is available via Git:

```
$ git clone git://genometools.org/gttestdata.git
```

The location of the `gttestdata` repository must be given when running the testsuite (see below). If it is not given, make sure that test which depend on large test data are disabled (e.g. by placing them in an '`if $gttestdata`' clause).

9.2.3 Running the testsuite

A comprehensive *GenomeTools* test run, containing both the unit tests and the tool tests, can be initiated by issuing `make test` in the *GenomeTools* directory. The following `make` switches influence the test runs:

- `memcheck=yes` enables memory access checking via *valgrind*,
- `testthreads=<n>` enables multithreaded testing with `<n>` threads in parallel to speed up test runs,
- `gttestdata=<path>` tells the test suite to look for large test data in `<path>`. This must be where a copy of the `gttestdata` repository is installed.

The tool tests can also be run using the `testsuite/testsuite.rb` script. Use the `-keywords <keywords>` parameter to only run these tests tagged with the given keywords. OR and AND operators can be used to specify the tests in a more detailed way. The `-select <n>` parameter can be used to run only the one test with number `n`, and the `-threads <n>` will run the testsuite with `n` threads in parallel.

9.3 Header inclusion dependencies

Often function prototypes in the *GenomeTools* header files use types declared in another header files. By mistake, it is possible to forget `#include`'ing the header files where the type is defined in the header using it. Note that this problem may never surface if the forgotten header is included in every C source file which includes the header file with the missing include statement. To address this, the script `scripts/src_check_header.rb` tries to include each header file given as a command line argument by itself in a C file and compile it. If dependencies are missing, the check will abort and output the compiler error message so the problem can be fixed.

9.4 Debug symbols

Compilation with debug symbols is enabled by default. To make sure that line numbers are correct when using a debugger, e.g. `gdb`, use the `opt=no` option in the `make` call to disable compile-time optimization. The `opt` option is enabled by default.

9.5 Profiling

To enable the generation of profiling output in the compiled binaries, use the `prof=yes` option in the `make` call. The `prof` option is disabled by default. Enabling this option makes the *GenomeTools* binary create a `gmon.out` file during each run, which can then be used for analysis using *gprof*[2].

9.6 Logging

Use the `gt_log_*()` functions in `core/log_api.h` to log debug messages to the screen or files. Output of debugging information defined using these functions can then be enabled or disabled via the `-debug` option of the `gt` binary. That is, to run tool `mytool` with debug output enabled, run

```
$ gt -debug mytool
```

10 Additional `make` parameters

10.1 Additional targets

Simply running *make* builds both the `gt` executable and the *GenomeTools* shared library. However, there are also other targets (besides the ones mentioned in the respective sections above) which can be built using the *GenomeTools* Makefile:

- `docs`, which builds API documentation as web pages in `www` and as LaTeX source in `doc/`,
- `manuals`, which, in addition to the files created by `docs`, also creates manuals for some published tools in *GenomeTools* (see `doc/manuals`),
- `install`, which installs the compiled *GenomeTools* binaries, libraries and headers into the directory specified by the `prefix=<path>` option,
- `dist`, which creates a tarball with a binary *GenomeTools* distribution, which will then reside in the `dist/` subdirectory of the *GenomeTools* root,

[2] See `http://sourceware.org/binutils/docs/gprof/index.html` for further information.

- `srcdist`, which creates a tarball with a source *GenomeTools* distribution, which will then reside in the working directory,

- `spgt`, which checks selected files in the `core/` and `match/` subdirectories using the splint static checker[3], using the rule set `testdata/SKsplintoptions`,

- `clean`, which removes all files created during the build and test processes, except the `lib` and `bin` directories, and

- `cleanup`, which even removes these.

10.2 Additional options

There are additional `make` options which are also mentioned in the README file and which influence how the *GenomeTools* binaries are built:

- Use `amalgamation=yes` to compile *GenomeTools* as an *amalgamation*. That means, all *GenomeTools* C source files are concatenated into a big source file, which is then compiled. This approach allows the compiler to perform more extensive optimizations during the compilation and may result in better performance. It is encouraged to check regularly whether compiling *GenomeTools* as an amalgamation still works, as name clashes in static functions can sometimes occur which compile fine when in separate files, but lead to errors in the amalgamation. This option is disabled by default.

- Use `errorcheck=no` to make the compilation process not stop when a warning is encountered. This option should only be used if necessary (e.g. when building *GenomeTools* on Windows). This option is enabled by default.

- Use `cairo=no` to disable Cairo support in the *AnnotationSketch* component of *GenomeTools*. This is useful on systems on which there is no Cairo library present, and *AnnotationSketch* is not needed. This option is enabled by default.

- The option `sharedlib=no` disables building of a *GenomeTools* shared library. This option is enabled by default.

- The option `static=yes` tries to link all dependencies of *GenomeTools* statically. This option is disabled by default.

- Use `curses=no` to disable ncurses support. Useful when there is no ncurses library on the target system. This option is enabled by default.

11 Contributing code

For *GenomeTools* development, we use the distributed versioning system Git[4] to track changes and exchange new code. Thus a Git repository is necessary to both:

[3] `http://www.splint.org`

[4] For a good introduction to the use of the Git software itself, see the Git web site (`http://git-scm.com`) or read the following guide: Travis Swicegood. *Pragmatic Version Control Using Git*. Pragmatic Bookshelf, ISBN 1934356158. We *strongly* encourage future *GenomeTools* developers to familiarize themselves with Git before developing with the intent of submission!

- obtain the latest development version of the *GenomeTools*, and
- contribute to the *GenomeTools* by submitting new code to the maintainers.

Be aware that, in this guide, we will not explain Git basic concepts, or how individual Git commands work in detail. Instead, we will shortly state what strategy is most effective when working in *GenomeTools* development.

11.1 Getting started

To get started with *GenomeTools* development, we recommend the following:

1. Install the Git version control system.
2. Read the Git documentation.
3. Clone the *GenomeTools* Git repository with:

   ```
   $ git clone git://genometools.org/genometools.git
   ```

4. Start hacking on your own feature branch:

   ```
   $ cd genometools
   $ git checkout -b my_feature_branch_name
   ```

5. Have fun!

11.2 Basic Git configuration

Please set your username and email address correctly. If unconfigured, they are often based on the hostname of the workstation where a commit is done. This may not be – and almost never is – correct in typical development environments (i.e. `user@workstation.zbh.uni-hamburg.de` instead of `user@maildomain.org`).
Use the `git config` commands while in your *GenomeTools* Git repository to set them to a correct value:

```
$ git config user.name "Hans␣Mustermann"
$ git config user.email "mustermann@maildomain.org"
```

11.3 Tips for successful source management

- Develop each major feature or try out bigger changes in a separate branch (the so-called *feature branch*) dedicated only to that aspect. That makes it easier to combine or discard branches later on, without having to meddle with individual commits too much if something goes wrong. Creating, merging and deleting branches is cheap in Git!
- Always leave your *master* branch untouched so code pulled from upstream (e.g. the official *GenomeTools* repository) does not get merged by accident.
- Branch off new feature branches from the *master* branch only. That makes it easy to chain branches later via `git rebase` in any order.

- Try to keep commits atomic. Every commit should either add a single feature or fix a single bug. That makes two things easier:

  1. Locating the exact commit which introduces a bug, e.g. using `git bisect`. If there are too many changes in one commit, bugs become more tedious to track down.
  2. Reverting single commits if new features introduce bugs.

  If you made several incomplete commits and want to reorder or combine them into one afterwards, use interactive rebasing via `git rebase -i`[5].

- Needless to say, every commit should compile cleanly. Again, bisecting can become very tedious if the code has to be fixed at each stop to get it to even compile.

- Try to avoid merges with upstream as much as possible to keep the history linear. Instead of merging, rebase your feature branches regularly against an updated *master*:

  ```
  $ git branch
  * mybranch
    master
  $ git checkout master
  $ git pull origin
  <...>
  $ git checkout mybranch
  $ git rebase master
  <...>
  ```

- In the first line of the commit message, give a short description of the change contained in the commit. Please use active, present tense, e.g. “add feature X” or “allow X to do Y”. Commit messages for commits that touch scripting language bindings should be prefixed with the language in question, e.g. “gtpython: add bindings for GtFoo class”.

- For bugfixes for bugs also listed in the *GenomeTools* Lighthouse bug tracking web app[6], commit messages can be used to update the ticket status. See the Lighthouse documentation[7] for details.

11.4 Submission of contributions

This section describes how to get your contributions noticed, reviewed and integrated into the main *GenomeTools* codebase.

11.4.1 Source code submission

To get your source code to be considered for inclusion into the *GenomeTools* official source tree, first you need a Git repository in a publicly accessible location. That may be a free public account on github (`http://www.github.com`, Gitorious (`http://gitorious.org`), etc., or a directory on a filesystem accessible by one of the core developers.

Typically the *GenomeTools* maintainers pull from a branch provided in this location and specified by you. This is called a *pull request*. Please specify the exact location to the repository as well as the branch name. A pull request can be sent to

[5] See `http://book.git-scm.com/4_interactive_rebasing.html` for an explanation.

[6] `http://genometools.lighthouseapp.com`

[7] `http://help.lighthouseapp.com/faqs/ticket-workflow/ticket-keyword-updates`

steinbiss@zbh.uni-hamburg.de

or

gremme@gmail.com.

Important: Always rebase your feature branch against the current official *GenomeTools* master before requesting a pull (see above). Also, please check whether your code compiles cleanly, even with the `amalgamation=yes` and `assert=no` parameters enabled which may influence compilation success. Small changes can also be submitted as patches which must be applicable using `git apply`. You will then be credited in the commit message. Such patches can be created conveniently – ready to be sent via mail – via git only using `git format-patch` or `git diff`.

11.4.2 Test data submission

For submissions to the `gttestdata` repository, the same rules apply as for source code. Please provide a repository from which to pull a branch which has been rebased against the current `gttestdata` master before. Before adding any more test data to the repository, please make sure that the new data is absolutely necessary. That is, existing large sequence should be reused, for example when testing a sequence parser or the like.

11.4.3 Licensing

Note that the *GenomeTools* are free software, i.e. an open-source project. All code distributed with the *GenomeTools* is published under the ICS license, which can be viewed at http://genometools.org/license.html. Submission of code for inclusion into the *GenomeTools* implies your permission to publish your code under this license. We will not accept contributions lacking proper copyright information at the top of each source file (see 5.5)!