Software Developer Candidate Exercise

GoodMorning.com Inc.

May 12, 2022

Contents

1 Background

Congratulations on reaching the second stage of the GoodMorning.com software developer candidate interview process. For this stage you are asked to complete the seemingly simply task explained in section 2. The purpose of this exercise is to evaluate you on the following abilities:

- Taking objectives and requirements, and coherently translating them into quality code implementations via effective object oriented design and planning.
- Developing collaborative implementations that maximize the success of your teammates when working with said implementations.
- Solving challenging programmatic problems with implementations that optimize for both readability, elegance, and computational efficiency.
- Reading and understanding existing code implementations.
- Reading, understanding and following written instructions and technical documentation.
- Writing technical documentation.
- Using (and figuring out how to use) some arbitrary development tools.

Anything that is not made clear by this document is a creative freedom you have in the approach you take. Although there is no "right answer" with respect to to this freedom, you will nonetheless be evaluated on the choices you make.

You are strongly encouraged to review this document in its entirety prior to any sort of execution. It will inform not just what would be expected of you at GoodMorning.com, but also what kind of development practises you yourself can expect to see here.

For the purposes of this exercise assume you are a developer working for an e-commerce SaaS company. You are currently assigned to the development team behind the Astiri product which is some sort of network request processing platform. Although you have plenty of creative freedom in the development approaches you use, the results you deliver are heavily bounded by established best practises and the standards outlined in section 3 on page 4. As far as programming language choice goes, you are bounded to PHP or C++. Be aware that if and when you reach the next stage of interview, the bulk of the interview will be questions on the work you submit for this exercise.

2 The Task

You have been tasked with making a minor enhancement to our Astiri product. The objectives and requirements motivating this enhancement are outlined in section 2.1 on the following page. These objectives and requirements should be the central motivating tenet behind all work you are doing. The specifics of the deliverable that you will be submitting for review are further clarified in section 2.2. Make sure you thoroughly review and understand all standards outlined in section 3 on the following page prior to starting. Pick either C++ or PHP as your language of choice.

2.1 Requirements

The product manager for our Astiri platform has discovered, via customer feedback, that a common objective of said customers is to gain a greater statistical understanding of request response times. In order to achieve these objectives it has been deemed that the current request handling class, shown in section 3.3 on page 6, should be expanded via inheritance to give us the required functionality.

Your job is thus to build, in-code document, and validate this child class, along with any other components you need to build, to the following requirements:

1. Instantiated objects must have a mechanism for retrieving the mean response time of all requests the object has processed for each separate URI. Type must be an appropriate real number with millisecond units.

2. Instantiated objects must have a mechanism for retrieving the standard deviation for the response time of all requests the object has processed for each separate URI. Type must be an appropriate real number with millisecond units.

3. Instantiated objects must have a mechanism for retrieving a *normalized* histogram for the response time of all requests it has processed for each separate URI.

   (a) The maximum number of bins per histogram should be provided during object instantiation. The actual number of bins should be as close as you deem practically possible to the maximum.

   (b) All response times must be captured in the histogram. No data can be truncated at the edges.

   (c) The histogram must contain no empty bins at the edges.

2.2 The Deliverable

The final deliverable that you submit for review must be assembled as such:

1. The original parent Request class must be included but unchanged.

2. Include some sort of high level design. This could be anything from a photo of a sketch done on a napkin to full fledged inheritance and collaboration diagrams. The intent is not to evaluate the tools and mechanisms you use for high level design but to show what deep thinking and planning you did *prior* to execution.

3. Class definition and implementation that meets the requirements outlined in section 2.1 on the preceding page and follows all coding standards itemized in section 3.1.
4. A command-line executable unit test that validates all behaviour for what you've built. Since response times are simulated in the Request class, leverage your understanding of that simulated behaviour to build your unit test.
5. If you opt for C++, include a cross-platform Makefile that will build everything including the unit test.
6. A basic markdown formatted README file that provides any context you deem relevant along with usage instructions for the unit test.
7. A fully standalone git repository that follows the etiquette outlined in section 3.2 on the following page and contains all the above plus anything needed to make it work.
8. Everything above should be archived into a tar.bz2 file and submitted as such.

Should you be unable to achieve all elements of the deliverable please submit your work as it will be fully evaluated nonetheless. Some parts are purposefully far more challenging than others.

3 Appendix

3.1 Coding Standards

All code of any kind should follow these guidelines:

1. All component (classes and functions) behaviour *must* be explicitly clear simply by reading and examining the public interface of said component. Ensure this by
   (a) leveraging maximally strong typing at interface points. Strong typing in this context means that *all* relevant type information should be immediately evident to a reader without needing to analyze behaviour at runtime.
   (b) documenting all interface points with in-code Javadoc style comments. See the Request class in section 3.3 on page 6 for examples.
   (c) minimizing interfaces. Public interfaces should expose only that which is *specified as required* of the component. Additional behaviour should be hidden in implementation when possible.
   (d) never exposing class data directly to a public interface.

   (e) documenting, at the interfaces, all implementation limitations (computational, complexity, error, etc) that are not immediately evident by the interface definitions.

2. Error management should use the exception approach with segregation of detection and handling. Seek fault tolerance in your code by emphasizing data validation at interface points whenever warranted and where strong typing is not sufficient.

3. Emphasize code that is both easy and pleasant to read. At *minimum* you should follow these styling conventions for uniformity:

   (a) Variables and functions are named using camelCase.

   (b) Classes, types and namespaces are named using CamelCase.

   (c) Member data should be named with a m_ prefix. The only time this isn't the case is for passive data structures.

   (d) Use spaces instead of tabs and indent at four spaces.

   (e) LF line terminations. No CRLF.

   (f) Cap line lengths to around 80 characters. Maximize readability by breaking up long lines along with artful use of whitespace.

4. All code should be completely free of development artifacts. Reading through your code should reinforce a *singular* design intention; no previous failed iterations. Every line should be purposeful and the support of your development practise is not an acceptable purpose. Leave us in such a state of awe that we want to read it to our kids before bed.

5. Any use of libraries outside the language standards are forbidden. This includes IDE specific functionality that is non-standard for the language.

3.2 Version Control Etiquette

Just as we must ensure the health of our code bases via coding standards we must also ensure that we can extract maximal value from its time dimension (version control history). We get this via our version control etiquette. This means that all individual commits on production branches *must* coincide with a single development operation (feature addition, bug fix, etc). No intermediate commits are permitted on said branches and no combining them into big commits. We should be able to roll back to any commit in the history and land at a deterministic and functional point. Practically this means that when merging to production branches you'll need to squash or rebase your development branch. Since each commit should encompass a single development operation, it should also contain a commit message that explains it fully. The following is an example of said commit messages containing relevant guidelines.

```
This is a brief summary of the commit in under 60 chars

This is an optional body paragraph of the commit message. In here we can
add additional information that couldn't fit in the summary title.
Notice that there is a blank line separating this paragraph from the
summary title above. This is required for formatting purposes. Also
notice that all lines are wrapped at 72 characters.

Now I've started a second paragraph. Notice that here as well we have a
a blank line separating this paragraph from the one above.
```

3.3 The Request Class

Included here is the Request class in both C++ and PHP.

3.3.1 C++

Request.hpp:

```cpp
#include <string>

/**
 * Resource request processing class
 *
 * Instantiations of this class do state based processing of resource requests.
 * To use, instantiate an object and call process() on a URI to get the response
 * data. Children of this class can augment functionality by overriding start()
 * and finish().
 */
class Request
{
private:
    /**
     * Simulate a delay that is specific to the URI in question
     *
     * Let's pretend that this function doesn't actually exist in this class
     *
     * @param [in] uri The URI of the request endpoint
     */
    static void simulateLatency(const std::string& uri);

protected:
    /**
     * Start processing the request in the child class
     *
     * @param [in] uri The URI of the request endpoint
     */
    virtual void start(const std::string& uri);

    /** Finish processing the request in the child class */
    virtual void finish();

public:
    /**
     * Process the request
     *
     * @param [in] uri The URI of the request endpoint
     * @return The response data
     */
    std::string process(const std::string& uri);
};
```

Request.cpp:

```
#include "Request.hpp"

#include <random>
#include <chrono>
#include <thread>
#include <map>
#include <algorithm>

void Request::simulateLatency(const std::string& uri)
{
    static std::map<std::string, std::normal_distribution<double>> distributions{
        {{"uri1"}, std::normal_distribution<double>(10000.0, 2500.0)},
        {{"uri2"}, std::normal_distribution<double>(20000.0, 7500.0)}};
    static std::normal_distribution<double> defaultDistribution(
            15000.0,
            5000.0);
    static std::random_device device;

    const auto it = distributions.find(uri);
    std::normal_distribution<double>& distribution(
            it == distributions.end() ? defaultDistribution : it->second);

    const auto responseTime = distribution(device);
    if(responseTime >= 1.0)
        std::this_thread::sleep_for(std::chrono::microseconds(
                    static_cast<std::chrono::microseconds::rep>(responseTime)));
}

std::string Request::process(const std::string& uri)
{
    start(uri);

    // Let's pretend the following line is doing something instead of just
    // simulating response latency
    simulateLatency(uri);

    finish();

    return "Sample␣response.";
}

void Request::start(const std::string& uri)
{
    // Base class version does nothing
}

void Request::finish()
{
    // Base class version does nothing
}
```

3.3.2 PHP

Request.php:

```
<?php

/**
 * Resource request processing class
 *
 * Instantiations of this class do state based processing of resource requests.
 * To use, instantiate an object and call process() on a URI to get the response
 * data. Children of this class can augment functionality by overriding start()
 * and finish().
 */
class Request
```

```
{
    /**
     * Means for simulated response latencies
     *
     * Let's pretend that this doesn't actually exist in this class. Unit are
     * microseconds.
     */
    private const MEANS = [
        'uri1' => 10000,
        'uri2' => 20000];

    /**
     * The default mean latency in microseconds
     *
     * Let's pretend that this doesn't actually exist in this class.
     */
    private const DEFAULT_MEAN = 15000;

    /**
     * Standard deviations for simulated response latencies
     *
     * Let's pretend that this doesn't actually exist in this class. Unit are
     * microseconds.
     */
    private const STDDEVS = [
        'uri1' => 2500,
        'uri2' => 7500];

    /**
     * The default standard deviation for latencies in microseconds
     *
     * Let's pretend that this doesn't actually exist in this class.
     */
    private const DEFAULT_STDDEV = 5000;

    /**
     * Simulate a delay that is specific to the URI in question
     *
     * Let's pretend that this function doesn't actually exist in this class
     *
     * @param string $uri The URI of the request endpoint
     */
    private static function simulateLatency(string $uri): void
    {
        // The following puts execution to sleep by a random amount of
        // microseconds. This amount is generated by transforming PHP's uniform
        // random number generation into Gaussian random number generation via
        // the Box-Muller transformation.
        $responseTime = round(
            sqrt(-2.0 * log(mt_rand(
                PHP_FLOAT_EPSILON*mt_getrandmax(),
                mt_getrandmax())/mt_getrandmax()))
            * (self::STDDEVS[$uri] ?? self::DEFAULT_STDDEV)
            * cos(2*pi()*mt_rand()/mt_getrandmax())
            + (self::MEANS[$uri] ?? self::DEFAULT_MEAN));
        if($responseTime >= 1)
            usleep($responseTime);
    }

    /**
     * Start processing the request in the child class
     *
     * @param string $uri The URI of the request endpoint
     */
    protected function start(string $uri): void
    {
        // Base class version does nothing
    }
    /** Finish processing the request in the child class */
    protected function finish(): void
    {
        // Base class version does nothing
    }

    /**
     * Process the request
     *
     * @param string $uri The URI of the request endpoint
     * @return string The response data
     */
    final public function process(string $uri): string
    {
        $this->start($uri);

        // Let's pretend the following line is doing something instead of just
        // simulating response latency
        self::simulateLatency($uri);

        $this->finish();

        return 'Sample␣response.';
    }
}
```