Final Project

Sabbir Ahmed

December 8, 2021

Contents

1 Introduction

In software engineering, design patterns are general, reusable solutions to commonly occurring problems [1]. It is generally considered good practice to integrate design patterns into software products, especially large projects, since it allows the developers to focus their time and attention towards specific implementations. The purpose of this draft is to introduce an open-source project and present analysis on the software patterns within the implementation.

2 Structural Simulation Toolkit (SST)

The software that is being focused on in the final project is Structural Simulation Toolkit (SST). It is a simulation framework that prioritizes high performance computing (HPC) models [2]. SST provides the user with a fully modular design in a parallel simulation environment based on MPI. The SST library can be imported in a C++ script to be executed as a model by a custom interpreter provided by SST. Several prebuilt models, known as SST Elements, have been implemented for frequently used simulation subsystems.

Due to SST being a large scale project with many stable extensions implemented for its kernel, the scope of the project will be limited to specific sections of the core repository. The repository is hosted on GitHub [3]. The source files that will be analyzed reside in `src/sst/core/`.

2.1 Project structure

This section describes, in a high level overview, the structure of SST's code base. Analysis of the layout will assist in understanding the various design patterns that are present or proposed for the project.

SST is structured as a library that is to be imported by the Client. The library implements and supplies its own `main` function, which restricts the Client from creating an entry point. In order to utilize the library, the Client must create derived classes to be executed with the command line tools provided by SST. The source files are compiled with the library using any popular C++ compilers that support MPI. The compiled objects can be executed by the provided SST executables that wrap the `mpirun` command. The Client is also required to provide accompanying Python scripts to provide driver functions with the desired parameters.

The following is a typical project layout using SST:

```
src
├─ parent.cpp
├─ model.cpp
├─ CMakeLists.txt
├─ run.py
└─ docs/
```

1. `CMakeLists.txt` is responsible for linking the files with SST and compiling the shared objects with a C++ compiler

2. `run.py` is a required Python script that has to import the library into its interpreter to be executed by the provided executables. A typical method to run the user's model in the SST framework is `sst run.py`.

3 Software Patterns Present in SST

The following patterns can be observed to have been already implemented in the project:

1. Abstract factory pattern
2. Factory method pattern
3. Singleton pattern
4. Strategy pattern

Other patterns are present in the project, such as C++ idioms (Include Guard Macro, enable_if, etc.)

3.1 Abstract Factory/Factory Method

The abstract factory and factory method patterns are present in the `SST::Factory` class. In the repository, the class can be located at `factory.h`. In the repository, it is used to create several concrete classes, including `Component` and `Module` objects. The class also provides templated variadic methods to create concrete classes of generic classes, such as

```
// src/sst/core/factory.h
/*
 * General function to create a given base class.
 *
 * @param type
 * @param params
 * @param args Constructor arguments
 */
template<class Base, class ... CtorArgs>
Base* Create(const std::string& type, CtorArgs&& ... args)
```

3.2 Singleton

The singleton pattern is present in the `SST::Factory` class. In the repository, the class can be located at `factory.h`. The class is used to instantiate other concrete simulation classes. SST requires simulation objects to be synchronized throughout the kernel, especially since they can be running on a distributed system where race conditions can become major issues. The software forces these simulation objects to be singletons.

3.3 Strategy pattern

The strategy pattern is present in the `SST::Core::Serialization::serializer` class. The class is implemented throughout multiple files in `serialization`, where it is overloaded in the files with various parameter types, with all the various versions of the class simply overloading the function call operator (`operator()`).

4 Recommended Software Patterns in SST

The following patterns can be considered appropriate to implement in the project:

1. Façade pattern
2. Interpreter pattern

4.1 Façade pattern

The current method for a Client to interface the library is to create a derived class of Component and override its methods. While this approach provides extensive control over the functionality of crucial methods such as `void setup(unsigned int)`, `void finish(unsigned int)` and `bool tick(SST::Cycle_t)`, it requires the Client to have extensive knowledge of the subsystems in the framework. The aforementioned methods, if overridden by the Client, must be implemented properly for the model and the simulation to be functional.

The following listing is an interface of a simple Component that simulates a primitive full adder hardware unit.

```
#include <sst/core/component.h>
#include <sst/core/interfaces/stringEvent.h>
#include <sst/core/link.h>

class FullAdder : public SST::Component {
public:
    // register and manually configure each of the SST::Links
    // to their corresponding event handlers
    FullAdder(SST::ComponentId_t id, SST::Params& params);

    // implement logic for the model when it is being loaded into
    // the simulation
    void setup() override;

    // implement logic for the model when it is being unloaded from
    // the simulation
    void finish() override;

    // implement logic for the model on every clock cycle in
    // the simulation
    bool tick(SST::Cycle_t cycle);

    // event handlers for all the member SST::Link attributes
    void handle_opand1(SST::Event* event);
    void handle_opand2(SST::Event* event);
    void handle_cin(SST::Event* event);

    // register the component
    SST_ELI_REGISTER_COMPONENT(
        FullAdder, // class
        "fulladder", // element library
        "fulladder", // component
        SST_ELI_ELEMENT_VERSION(1, 0, 0),
```

```
        "SST parent model",
        COMPONENT_CATEGORY_UNCATEGORIZED)

    // port name, description, event type
    SST_ELI_DOCUMENT_PORTS(
        {"opand1", "Operand 1", {"sst.Interfaces.StringEvent"}},
        {"opand2", "Operand 2", {"sst.Interfaces.StringEvent"}},
        {"cin", "Carry-in", {"sst.Interfaces.StringEvent"}},
        {"sum", "Sum", {"sst.Interfaces.StringEvent"}},
        {"cout", "Carry-out", {"sst.Interfaces.StringEvent"}})

private:
    // SST parameters
    std::string clock;

    // SST links
    SST::Link *opand1_link, *opand2_link, *cin_link,
        *sum_link, *cout_link;

    // other attributes
    std::string opand1, opand2, cin;
    SST::Output output;
};
```

This Component is a relatively simple example of a model that can be simulated in the SST framework. The hardware logic for the full adder will be implemented in the tick function, where the output values (`sum` and `cout`) are evaluated using the member attributes `opand1`, `opand2`, and `cin` after they are processed by their corresponding handlers.

Exposing all the complexity of the base methods to the Client can lead to many potential issues. One way to reduce the chances of such issues is to abstract away the steps and methods from the Client using a Facade design pattern. The library, in its current state, does not provide a method to call any of the constructors of the Simulation objects, such as Components and SubComponents. Execution of such objects is done through various command line tools. Even testing of the classes appear to be done through external tools and Python interpreters, which compare the outputs to the expected outputs rather than using asserts.

The following listing is a potential interface that may be possible with the integration of a Facade object into the project.

```
#include <sst/core/facade.h>

bool customTickFunc(unsigned int cycle) {
    // do something
}

int main(int argc, char* argv[]) {
    SST::Facade* facade = new SST::Facade(argc, argv);
    SST::Component* component = facade->getComponent();

    component->register(
        FullAdder, // class
        "fulladder", // element library
        "fulladder", // component
        SST_ELI_ELEMENT_VERSION(1, 0, 0),
```

```
        "SST parent model",
        COMPONENT_CATEGORY_UNCATEGORIZED);
    component->registerStringEventPort("opand1", "Operand 1");
    ...

    component->overrideTick(&customTickFunc);
    component->setMPIRank(0);
    component->run();

    delete component;
    delete facade;

    return 0;
}
```

4.2 Interpreter pattern

Appendices

A Full Adder Hardware Design

The contents...

References

[1] Design patterns. https://sourcemaking.com/design_patterns.

[2] The structural simulation toolkit. http://sst-simulator.org/.

[3] sstsimulator/sst-core. https://github.com/sstsimulator/sst-core.