Final Project

Sabbir Ahmed

December 10, 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 provides a high level overview of 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. Factory method pattern
2. Singleton pattern
3. Prototype pattern

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

3.1 Abstract Factory Pattern

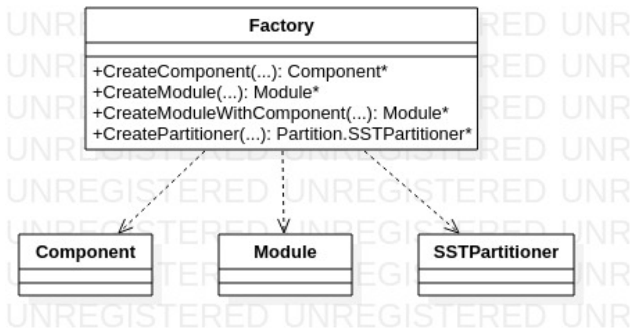
The abstract factory pattern is present in the `SST::Factory` class. In the repository, the class can be located at `factory.h`. The class is used to create several concrete products, including `Component` and `Module` objects.

The following excerpt lists the methods of the class following the typical steps dictated by the pattern.

Listing 1: Factory Implementing the Abstract Factory Pattern

```
// src/sst/core/factory.h
Component* CreateComponent(...);
Module* CreateModule(...);
Module* CreateModuleWithComponent(...);
Partition::SSTPartitioner* CreatePartitioner(...);
```

Figure 1: Example of a parametric plot $(\sin(x), \cos(x), x)$


Factory
+CreateComponent(...): Component*
+CreateModule(...): Module*
+CreateModuleWithComponent(...): Module*
+CreatePartitioner(...): Partition.SSTPartitioner*
Component
Module
SSTPartitioner
UNREGISTERED

3.2 Singleton Pattern

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.

The following listing consolidates all instances of the class following the typical steps dictated by the pattern.

Listing 2: Factory Implementing the Singleton Pattern

```cpp
// src/sst/core/factory.h
class Factory {
    ...
    Factory(const std::string& searchPaths);
    ~Factory();
    static Factory* instance;
    ...
}

// src/sst/core/factory.cc
Factory* Factory::instance = nullptr;

Factory::Factory(...) {
    if (instance) {
        out.fatal(CALL_INFO, 1, "Already initialized a factory.\n");
    }
    instance = this;
    ...
}
```

Although SST is primarily intended for multi-threaded applications, the Singleton class does not utilize any locks to account for the potential issues imposed by concurrency. Locks do exist in abundance throughout the project and within the class itself, but not when checking for instances of itself in other threads. The `Factory` class is responsible for creating SST Components, SubComponents, Modules, etc. as models for the simulation. It appears that the Singleton is instantiated by `mpirun` as a single thread which spawns the other processes after it completes analysis of the configuration options. The intent of the pattern is still preserved, although with the aforementioned assumptions that the executable instantiates it with a single thread.

The safety of the Singleton can be improved with the usage of mutexes to implement a double-checked locking [4], which is cited as an efficient method for implementing lazy initialization in a multi-threaded environment.

3.3 Prototype Pattern

The Prototype pattern is present in the project, although in a very limited manner. Select `SST::Core::Interfaces` classes implement a `clone` method to provide the ability to copy instances of themselves. The `StringEvent` class provides a shallow copy of itself to instantiate its ConcretePrototype, while `SimpleNetwork::Request` performs a deep copy.

The following listings demonstrate the instances of the project following the typical steps dictated by the pattern:

Listing 3: StringEvent Implementing the Prototype Pattern

```cpp
class StringEvent : public SST::Event, ... {
    ...
    virtual Event* clone() override {
        return new StringEvent(*this);
    }
    ...
}
```

Listing 4: SimpleNetwork::Request Implementing the Prototype Pattern

```
class SimpleNetwork : public SubComponent {
    class Request : public ... {
        ...
        inline Request* clone() {
            Request* req = new Request(*this);
            // Copy constructor only makes a shallow copy, need to
            // clone the event.
            if (payload != nullptr) {
                req->payload = payload->clone();
            }
            return req;
        }
        ...
    }
    ...
}
```

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.

Listing 5: Example Interface of an SST Component Model

```
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.

Listing 6: Potential Implementation of Facade

```
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;
```

4.2 Interpreter Pattern

SST requires external Python scripts to configure and run the Client models. The configuration file provides a vast amount of options; from setting the duration of the simulation to providing custom parameters to the models. The configuration file is imported by the provided executables and parsed by the `SST::Core::Model::Python` classes. The classes map the parameterized method calls made in the configuration file to the implemented class methods.

This method of interpreting the configuration file to generate SST Components for its simulation requires the usage of a Python interpreter. This external dependency potentially adds unnecessary overhead that can be circumvented by integrating the Interpreter pattern into the project. Adding an AbstractExpression class in native C++ will allow the project to not have to rely on an external framework and language to run its simulations.

Instead of a Python script, the simulation configuration can be represented in an XML format. XML parsing is a fairly common task in commercial software that is often achieved by external libraries. Configuration options in SST may require a very simple parser. The AbstractExpression class will perform a similar role to the current `SST::Core::Model::Python` classes and create SST Components for the simulation.

The following listings are an example of the potential translation the method can bring to parsing configurations:

Listing 7: Example SST Configuration File

```python
# run.py
import sst

sst.setProgramOption("stopAtCycle", "5s")

full_add_0 = sst.Component("Full Adder 0", "fulladder.fulladder")
full_add_0.addParams({"clock": "1Hz"})

full_add_1 = sst.Component("Full Adder 1", "fulladder.fulladder")
full_add_1.addParams({"clock": "1Hz"})

full_add_2 = sst.Component("Full Adder 2", "fulladder.fulladder")
full_add_2.addParams({"clock": "1Hz"})

full_add_3 = sst.Component("Full Adder 3", "fulladder.fulladder")
full_add_3.addParams({"clock": "1Hz"})

sst.Link("cout0").connect(
    (full_add_0, "cout0", "1ps"),
    (full_add_1, "cin1", "1ps")
)
sst.Link("cout1").connect(
    (full_add_1, "cout1", "1ps"),
    (full_add_2, "cin2", "1ps")
)
sst.Link("cout2").connect(
    (full_add_2, "cout2", "1ps"),
    (full_add_3, "cin3", "1ps")
)
```

Listing 8: Potential Implementation of an SST Configuration File

```xml
<? xml version="1.0" encoding="UTF-8"?>
<config>
    <setProgramOption stopAtCycle="5s"/><setProgramOption/>
    <component id="full_add_0" name="Full Adder 0" class="fulladder"/>
        <param key="clock" value="1Hz"/><param/>
        <link from="cout0" to="full_add_1" port="cin1"/></link>
    <component/>
    <component id="full_add_1" name="Full Adder 1" class="fulladder"/>
        <param key="clock" value="1Hz"/><param/>
        <link from="cout1" to="full_add_2" port="cin2"/></link>
    <component/>
    <component id="full_add_2" name="Full Adder 2" class="fulladder"/>
        <param key="clock" value="1Hz"/><param/>
        <link from="cout2" to="full_add_3" port="cin3"/></link>
    <component/>
    <component id="full_add_3" name="Full Adder 3" class="fulladder"/>
        <param key="clock" value="1Hz"/><param/>
    <component/>
</config>
```

4.3 Other Minor Idioms

The project can be found with several `goto` statements within the code base. The use of this unconditional jump is highly discouraged in the C++ community, although its usage is not considered a part of any idioms. Regardless, the project would benefit from discarding these statements and replacing them with proper conditional jumps for increased clarity and understanding of the project.

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`.

[4] The "double-checked locking is broken" declaration. `https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`.