605.707 Software Patterns
Final Project

Sabbir Ahmed

December 14, 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 paper is to introduce an open-source project and present an analysis of 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 are located in `src/sst/core/`.

2.1 Project structure

This section provides a high-level overview of the structure of SST's codebase. 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 `main` function, which restricts the Client from creating an entry point. 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 are additional files that are required to run a single-file SST model.

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.

Listing 1: A typical method to run the user's model in the SST framework

```
1 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

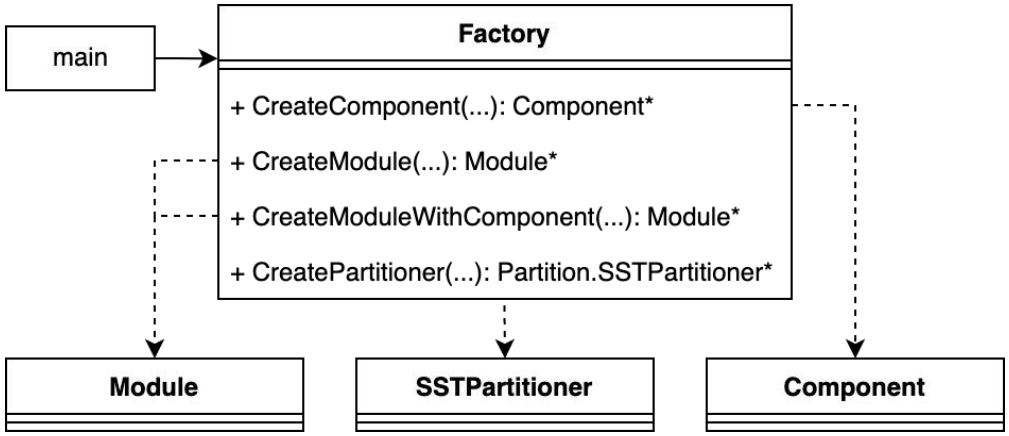
Other patterns are present in the project, such as C++ idioms (Include Guard Macro, enable_if, etc.)

3.1 Abstract Factory Pattern

The abstract factory pattern is present in the `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.

Figure 1 lists the methods of the class following the typical steps dictated by the pattern.

Figure 1: Factory Implementing the Abstract Factory Pattern



Each of the methods accept parameters to instantiate the corresponding ConcreteProducts. There are several Client classes that call these methods, including `Simulation`, `BaseComponent` and the `main` function itself.

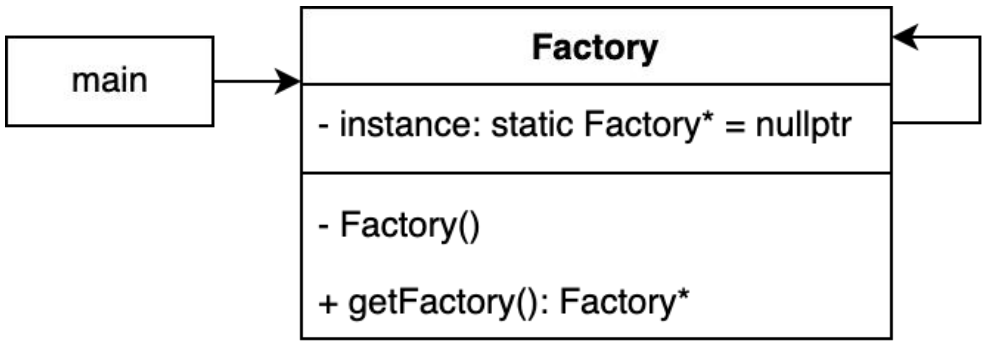
3.2 Singleton Pattern

The singleton pattern is present in the `Factory` and `Simulation.Simulation_impl` classes. In the repository, the classes can be located at `factory.h` and `Simulation_impl.h` respectively.

3.2.1 Factory

The `Factory` class is used to instantiate other concrete simulation classes. SST requires 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.

Figure 2: Factory Implementing the Singleton Pattern



Listing 2 demonstrates the `Factory` class following the typical steps dictated by the pattern.

Listing 2: Factory Implementing the Singleton Pattern
Files: src/sst/core/factory.h and src/sst/core/factory.cc

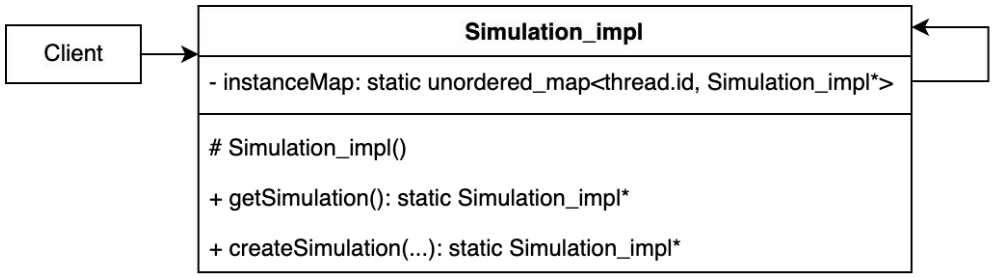
```
1  class Factory {
2    Factory(...);
3    ~Factory();
4    static Factory* instance;
5  }
6
7  Factory* Factory::instance = nullptr;
8
9  Factory::Factory(...) {
10   if (instance) {
11     ...
12     // exit 1
13   }
14   instance = this;
15 }
```

Although SST is primarily intended for multi-threaded applications, the `Factory` 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 the class 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 the 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.

3.2.2 Simulation_impl

Figure 3: Simulation_impl Implementing the Singleton Pattern



Listing 3 demonstrates the declaration of the other Singleton class, `Simulation_impl`.

Listing 3: Excerpt of Simulation_impl Interface
File: src/sst/core/simulation_impl.h

```
1  class Simulation_impl : public Simulation {
2  public:
3    static Simulation_impl* getSimulation();
4    static Simulation_impl* createSimulation();
5    static void shutdown();
6  protected:
7    Simulation_impl(...) {}
8  private:
9    static std::unordered_map<std::thread::id, Simulation_impl*> instanceMap;
10 }
```

Unlike `Factory`, this class is used by multiple classes in the framework which potentially may be on multiple threads. The `Simulation_impl` class does implement a relatively safer version of a Singleton instance by using a mutex.

Listing 4 demonstrates the `Simulation_impl` class following the typical steps dictated by the pattern, using a mutex to account for potential issues raised by multi-threaded processes.

Listing 4: Simulation_impl Implementing the Singleton Pattern
File: src/sst/core/simulation_impl.cc

```
1  std::unordered_map<std::thread::id, Simulation_impl*>
2    Simulation_impl::instanceMap;
3
4  Simulation_impl* getSimulation() {
5    return instanceMap.at(std::this_thread::get_id());
6  }
7
8  Simulation_Impl::createSimulation(...) {
9    std::thread::id tid = std::this_thread::get_id();
10   Simulation_impl* instance = new Simulation_impl(...);
11
12   std::lock_guard<std::mutex> lock(simulationMutex);
13   instanceMap[tid] = instance;
14   ...
15   return instance;
```

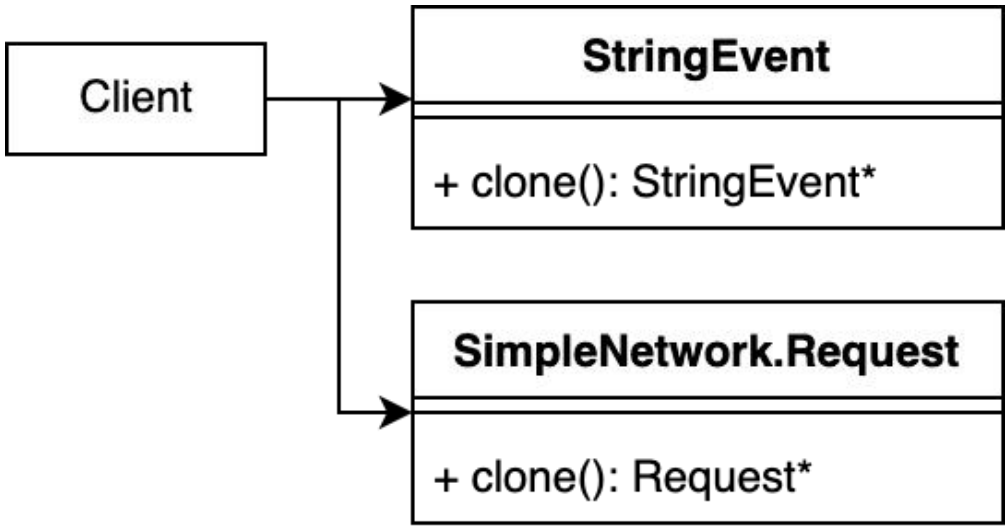
```
16 }
17
18 void Simulation_impl::shutdown() {
19   instanceMap.clear();
20 }
```

`Simulation_impl` appears to create an instance of itself on every thread and store them in the static unordered_map attribute, instanceMap. Each of the instances follows the restrictions enacted by the Singleton pattern. This method allows for a Singleton object on every thread and therefore the core intent of the pattern remains preserved. The map's contents are destroyed with the public `shutdown` method which is accessible by the main function, which runs on the primary thread.

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.

Figure 4: SST Classes Implementing the Prototype Pattern



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

Listing 5: StringEvent Implementing the Prototype Pattern
File: src/sst/core/interfaces/stringEvent.h

```
1 class StringEvent : public SST::Event, ... {
2   virtual Event* clone() override {
3     return new StringEvent(*this);
4   }
5 }
```

Listing 6: SimpleNetwork::Request Implementing the Prototype Pattern
File: src/sst/core/interfaces/simpleNetwork.h

```
class SimpleNetwork : public SubComponent {
  class Request : public ... {
    Request* clone() {
      Request* req = new Request(*this);
      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:

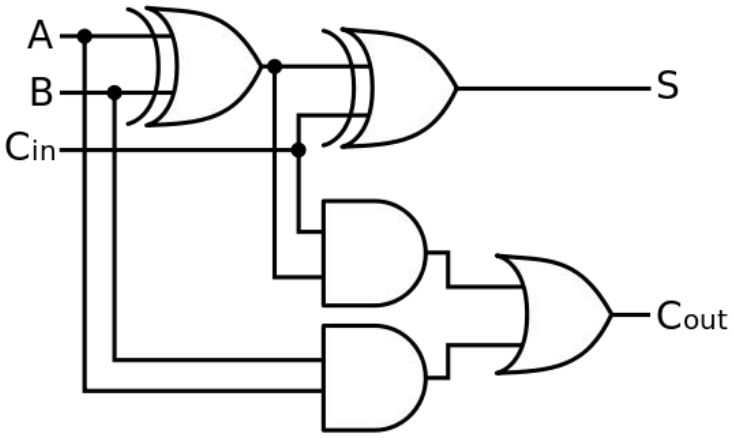
1. Façade pattern
2. Interpreter pattern

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 `setup(unsigned int):void`, `finish(unsigned int):void` and `tick(SST.Cycle_t):bool`, 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.

Listing 7 is an interface of a simple Component that simulates a primitive full adder hardware unit.

Figure 5: Circuit Design of a Simple Full Adder



Listing 7: Example Interface of an SST Component Model
File: FullAdder.hpp

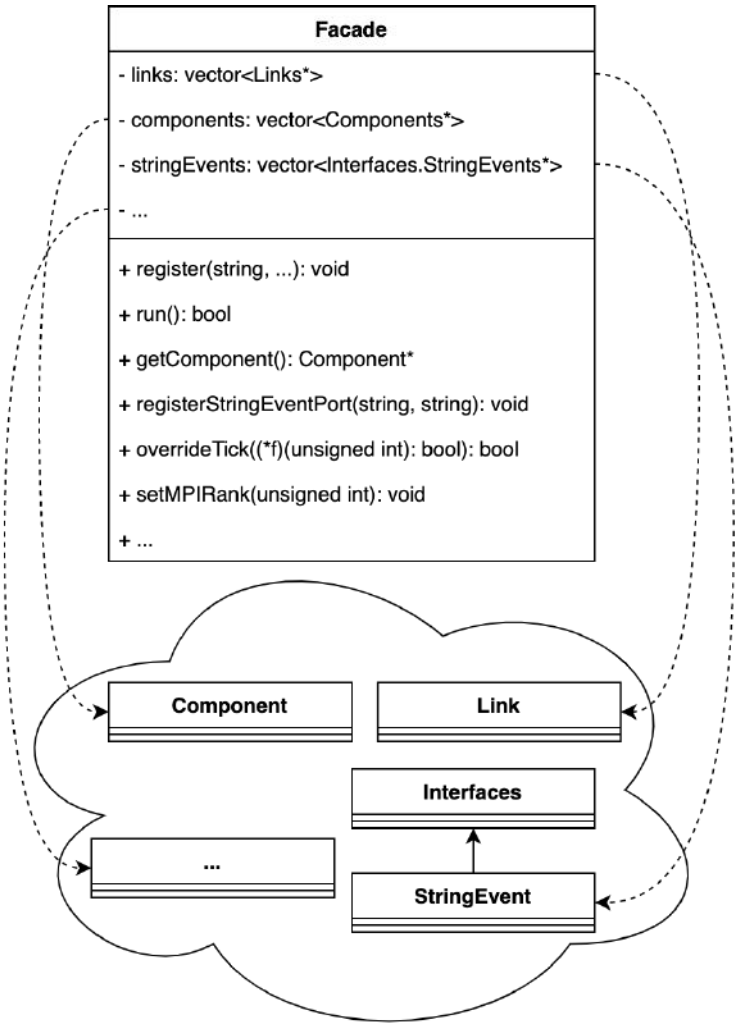
```
#include <sst/core/component.h>
#include <sst/core/interfaces/stringEvent.h>
#include <sst/core/link.h>

class FullAdder : public SST::Component {
  // SST parameters
  std::string clock;
  // SST links
  SST::Link *A_link, *B_link, *C_in_link, *S_link, *C_out_link;
  // other attributes
  std::string A, B, C_in;
  SST::Output output;
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_A(SST::Event* event);
  void handle_B(SST::Event* event);
  void handle_C_in(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 example model",
    COMPONENT_CATEGORY_UNCATEGORIZED)

  // port name, description, event type
  SST_ELI_DOCUMENT_PORTS(
    {"A", "Operand 1", {"sst.Interfaces.StringEvent"}},
    {"B", "Operand 2", {"sst.Interfaces.StringEvent"}},
    {"C_in", "Carry-in", {"sst.Interfaces.StringEvent"}},
    {"S", "Sum", {"sst.Interfaces.StringEvent"}},
    {"C_out", "Carry-out", {"sst.Interfaces.StringEvent"}})
};
```

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 (`S` and `C_out`) are evaluated using the member attributes `A`, `B`, and `C_in` 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 appears 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.

Figure 6: Potential Implementation of the Facade Pattern



Listing 8: Potential Implementation of Facade

```
SST::Facade* facade = new SST::Facade(argc, argv);
SST::Component* component = facade->getComponent();

component->register(
  "FullAdder", // class name
  "fulladder", // element library
  "fulladder", // component
  "1.0.0",
  "SST parent model");
component->registerStringEventPort("A", "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 `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 `Model.Python` classes and create SST Components for the simulation.

The following listing is an example of a configuration file for the preceding FullAdder model.

Listing 9: Example SST Configuration File
File: run.py

```
1  import sst
2
3  sst.setProgramOption("stopAtCycle", "5s")
4
5  full_add_0 = sst.Component("Full Adder 0", "fulladder.fulladder")
6  full_add_0.addParams({"clock": "1Hz"})
7
8  full_add_1 = sst.Component("Full Adder 1", "fulladder.fulladder")
9  full_add_1.addParams({"clock": "1Hz"})
10
11 full_add_2 = sst.Component("Full Adder 2", "fulladder.fulladder")
12 full_add_2.addParams({"clock": "1Hz"})
13
14 full_add_3 = sst.Component("Full Adder 3", "fulladder.fulladder")
15 full_add_3.addParams({"clock": "1Hz"})
16
17 sst.Link("C_out0").connect(
18     (full_add_0, "C_out0", "1ps"),
19     (full_add_1, "C_in1", "1ps")
20 )
21 sst.Link("C_out1").connect(
22     (full_add_1, "C_out1", "1ps"),
23     (full_add_2, "C_in2", "1ps")
24 )
25 sst.Link("C_out2").connect(
26     (full_add_2, "C_out2", "1ps"),
27     (full_add_3, "C_in3", "1ps")
28 )
```

The following listing is a potential translation of the preceding configuration file.

Listing 10: Potential Implementation of an SST Configuration File
File: run.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="C_out0" to="C_in1" dest="full_add_1"/></link>
  <component/>
  <component id="full_add_1" name="Full Adder 1" class="fulladder"/>
    <param key="clock" value="1Hz"/><param/>
    <link from="C_out1" to="C_in2" dest="full_add_2"/></link>
  <component/>
  <component id="full_add_2" name="Full Adder 2" class="fulladder"/>
    <param key="clock" value="1Hz"/><param/>
    <link from="C_out2" to="C_in3" dest="full_add_3"/></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 codebase. 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.

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`.