Design Description

Participants

Reece Karge (rkarge3)
Alexandr Dorofeyev (adorofeyev3)
Kimanii Daniel (kdaniel38)
Andre Pollard (apollard6)
Michael Tidwell (mtidwell3)

20 September 2014

Overview

The heated plate simulation program runs multiple implementation of the same basic algorithm and then displays the result to the user. This immediately brought up a difficult question: How to design a GUI that can run and display results from an unknown program? After researching design patterns, we found that there are many designs that use factorization to create a specific implementation, of a superclass or an interface, at run time. Some examples are Strategy Pattern, Template Method Pattern and Factory Pattern. After comparing these design patterns we chose to implement the Template Method Pattern. The “Template Method Pattern” defines a stub for an algorithm, deferring some implementation steps to subclasses. Template Method is mostly used when two or more implementations of a similar algorithm exist which is ideal for our Heated-Plate simulations where the basic algorithm remains but the structures used to house the values varies. Template Method Pattern is defined as a Behavior Pattern where the behaviour of our Heat-Plate algorithm varies. The responsibilities for implement this behavior is left to the subclasses and they do so by overriding the abstract methods in the parent class

ensuring each version will represent the same structure, even though the exact implementation may differ slightly.

Program characteristics

We built four (4) programs to study the variation of precision on the simulation of the heating plate and one GUI program to study usability and adaptability of the underlying design. Below we detailed the specifics of the programs:

Program	Total Lines of Code	Total Size in Bytes of Class File	Number of Classes	Average Number of Methods per Class	Average Number Attributes per Class	Number of Inter-Class Dependencies
Tpdahp	125	3,246	2	7	5	2
Tpfahp	109	3,095	2	7	5	2
Twfahp	111	3,340	2	7	5	2
Tpdohp	206	11,243	5	7	9	2
Gallhp	538	46,000	11	5.6	4.3	1

Program Naming Reference

Character Position	Symbol	Symbol Meaning
First	T	Text
First	G	Graphics
Second	p	Primitive data type to hold temperatures
Second	w	Wrapped data type to hold temperatures
Third	d	Double precision floating point computation
Third	f	Single precision floating point computation
Fouth	a	Array indexing used to access neighboring elements
Fouth	o	Object indexing used to access neighboring elements

Tpdahp - all computations are performed in double precision using a two-dimensional array of doubles to represent the plate

Tpfahp - Same as above except we used floating point calculations on an array of floats

Twfahp - Same as above except we encapsulate the float values in an object of Float within the array, this would require boxing and unboxing of values for calculations

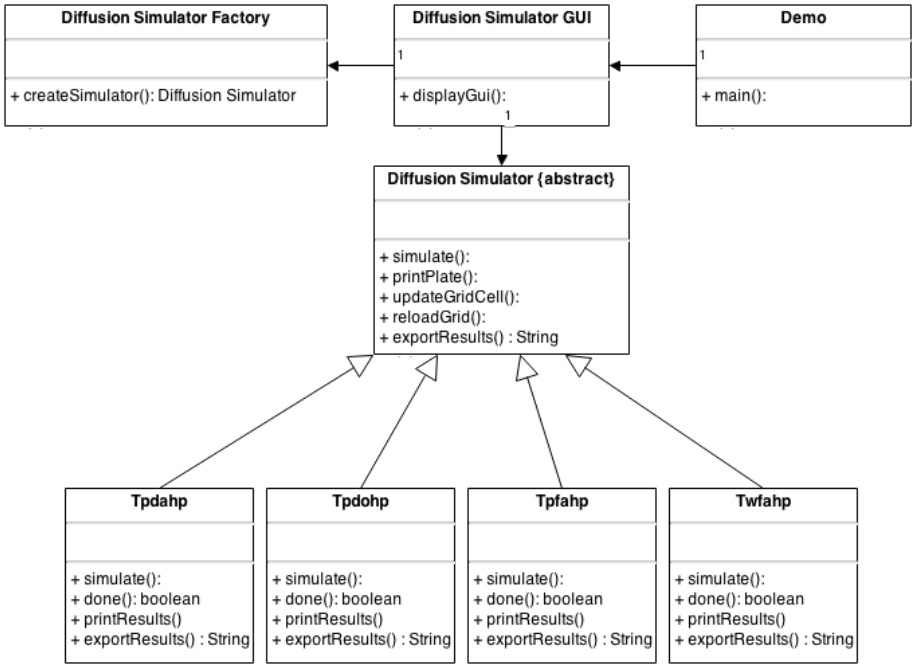
Twdohp - Same as above Instead, each lattice point was represented using an object that has attributes referring to each of its four neighboring lattice-point objects and a double variable for it's own temperature.

Gallhp - Graphic User Interface (GUI) that enabled the user to execute any of the four previous programs and to see a visualization of the results.

UML Class Model design diagram

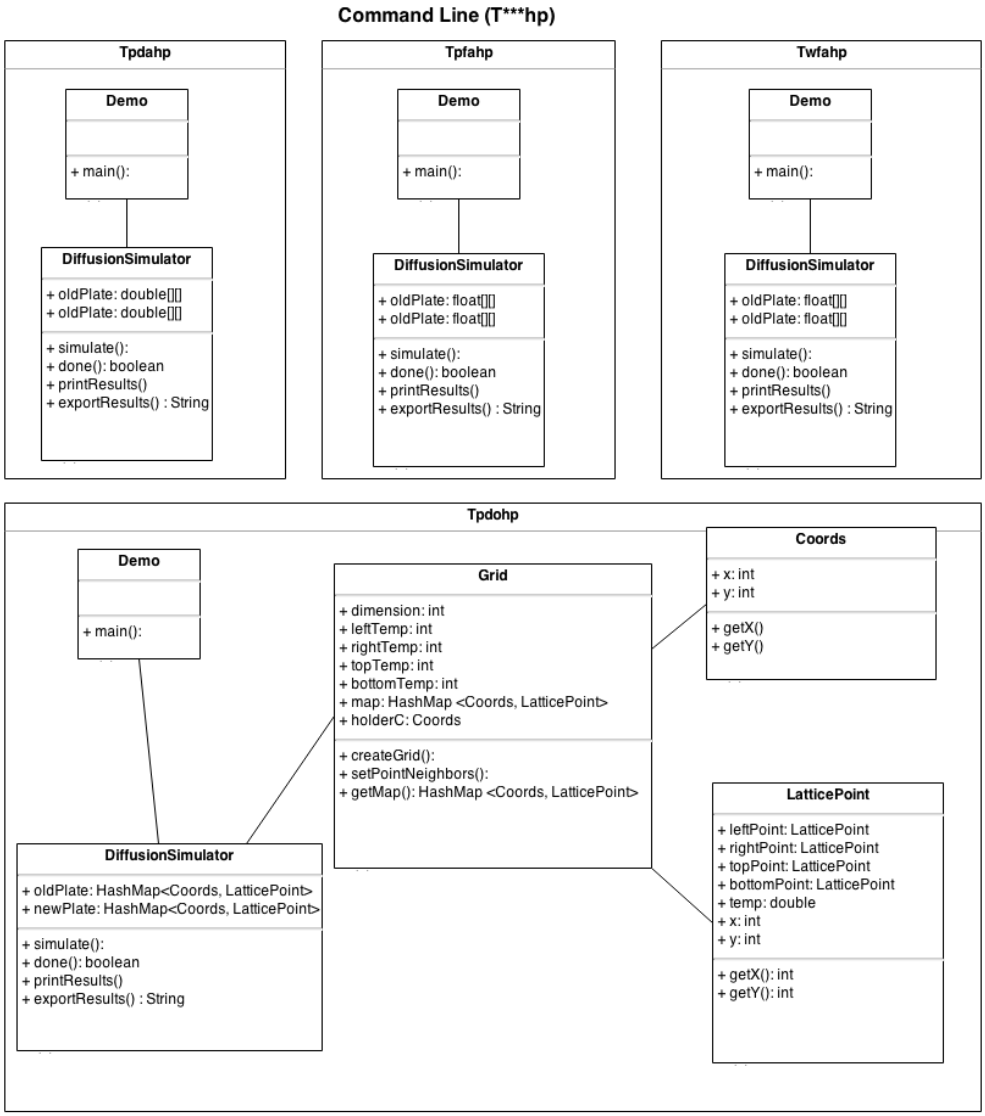
Gallhp:

Below is the UML for the GUI application. The separate programs (Tpdahp, Tpdohp, Tpfahp, Twfahp) is an abstraction of Diffusion Simulator.



T*hp:**

Below is the UML for the separate programs used to conduct the experiment. Each program is executed from the Demo stub and Demo instantiates the DiffusionSimulator class then calls it's various methods.



Static description

We followed an object oriented approach in the design of the programs, where classes were used as an organisational structure to group related segments of code into a single functioning unit. The major classes used in our design were: Demo, DiffusionSimulator, DiffusionSimulatorGUI, DiffusionSimulatorFactory, Tpdahp, Tpfahp, Twfahp, Tpdohp. Demo class was the main entry points for the programs.

The GUI was designed on proper object oriented principles, employing the use of abstractions and factorization. The GUI consisted of three main components - DiffusionSimulator, DiffusionSimulatorFactory, DiffusionSimulatorGUI and the four sub programs (Tpdahp, Tpfahp, Twfahp, Tpdohp). The DiffusionSimulator is an abstract class that is used to constrain the design of the four sub concrete classes (Tpdahp, Tpfahp, Twfahp, Tpdohp). The DiffusionSimulator class contains signatures of all the necessary methods for the experiment, and a constructor that creates the grid that will simulate the heated plate using the dimensions passed in. The DiffisionSimulatorFactory class employs the factory method priorily mentioned to create an instance of the requested sub program class. So it takes the “type” of program required, along with all the parameters for this program and it returns an instance of DiffusionSimulator which is an abstraction of the specific program requested. It also passes the necessary parameters to the program so it can setup the grid for the experiment. The calling application will only have to run the simulate method on the returned instance. The DiffusionSimulatorGUI housed the interactive components which allows the user to enter the necessary parameters for the experiment (temperatures, grid size and type of experiment) and execute the experiment. Below is a table of the classes and their dependencies:

Class Dependencies Table

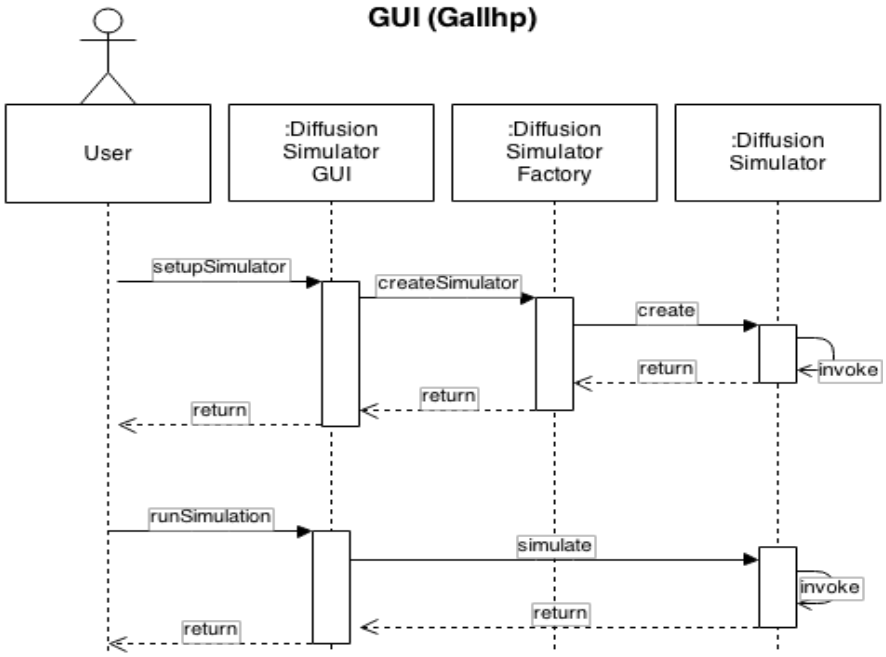
Packages	Major Classes	Role	Dependencies Type	Dependencies
T***hp	Demo	Program Entry	instantiates	DiffusionSimulator
T***hp	DiffusionSimulator	Concrete Class	instantiated by	Demo
Gallhp	Demo	Program Entry	calls	SwingUtilities
Gallhp	DiffusionSimulatorFactory	Factory	creates	DiffusionSimulator
Gallhp	DiffusionSimulator	Abstract Class	returns as result	Tpdahp
Gallhp	DiffusionSimulator	Abstract Class	returns as result	Tpdohp
Gallhp	DiffusionSimulator	Abstract Class	returns as result	Tpfahp
Gallhp	DiffusionSimulator	Abstract Class	returns as result	Twfahp

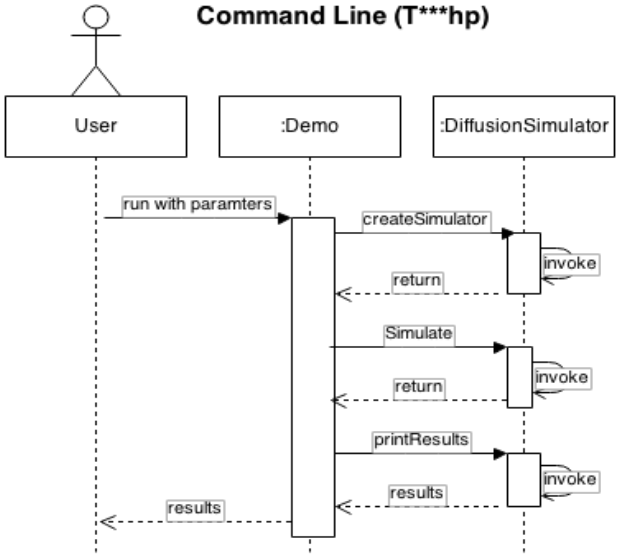
Class Dependency Description

Dependency Type	UML Stereotype	Description
calls	«call»	An operation in the source class invokes an operation in the target class
creates	«create»	The source class creates an instance of target class
instantiates	«instantiate»	An operation in the source class creates an instance of target class
uses as parameter		The source class contains an operation with a parameter of the target class
returns as result		The source class contains an operation that returns a result of target class
sends a message to	«send»	The source class contains an operation that sends a message of type target
refers to globally		The source class refers to target class, for example, to obtain static final values

Sequence diagrams

UML sequence diagrams illustrate the process flow from user to the system and how the user interacts with the system over time. It also depicts the objects and classes involved in the process and the sequence of messages exchanged between the objects needed to carry out the functionality of the experiment. Below are the UML for the GUI (Gallhp) and the command line (T***hp) programs.


GUI (Gallhp)
User
:Diffusion Simulator GUI
:Diffusion Simulator Factory
:Diffusion Simulator
setupSimulator
createSimulator
create
return
invoke
return
return
runSimulation
simulate
invoke
return
return


Command Line (T***hp)
User
:Demo
:DiffusionSimulator
run with paramters
createSimulator
invoke
return
Simulate
invoke
return
printResults
invoke
results
results

Dynamic description

For the command line implementations, the user may run the application through calling Demo and passing parameters for dimension, right temperature, left temperature, top temperature and bottom temperature, parameters - d,r,l,t,b respectively. The program will run and display the results as a grid on the screen. The program is executed at the command prompt as follows:

```
"java <packageName>.Demo -d # -l # -r # -t # -b #"
```

The `<packageName>` is replaced with the relevant program (Tpdahp, Tpdohp, Twfahp, Tpfahp) and the `#`'s are replaced with the relevant test values. The Demo class will verify the input parameters to ensure validity then instantiates the DiffusionSimulator class passing the necessary parameters to it. Demo then executes the simulate method to run the experiment followed by the printResults method, which would print the results of the experiment to screen.

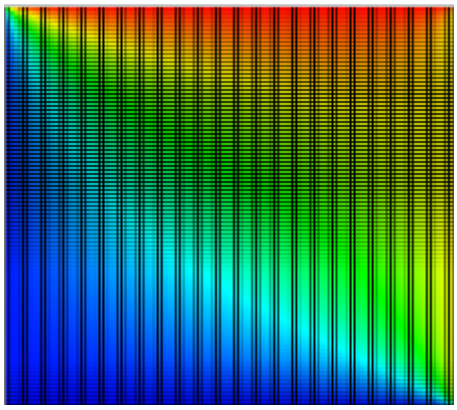
The GUI affords a more usual friendly experience to interact the various programs of the simulation. It allows a user to not only specify the input parameters for the simulation but also which simulator to run i.e Tpdahp, Tpdohp, Twfahp, Tpfahp. First a user would enter into text boxes the parameters for dimension, right, left, top and bottom temperatures, then select which simulator from the drop down list then press the "Setup Simulator" button(Step 1 in the GUI sequence diagram). After the setup completes, the user may then select "Run Simulation" button(Step 2 in the GUI sequence diagram) which will continuously display results as the algorithm runs. This is a very intuitive interface designed primarily to improve the user experience.

Low-level design considerations

When thinking about the design method to use to conduct the experiment we wanted a method that would adhere to the DRY principle (Don't repeat yourself). We realised that the sub programs were very similar in functionality and methods they made public, so we needed a design pattern that would allow us to swap functionality easily while not repeating code and allow us easy extensibility. With that in mind we discussed the Template and Strategy pattern. The choice between the Template Method Pattern and the Strategy Pattern was difficult and we eventually chose the Template Method Pattern. Both of these patterns gave us the ability to use a factory to decide which algorithm/program to run based on user input. They will also allow us to keep adding different algorithms/programs without having to update the GUI. But the one ability that the Template Method Pattern gives that the Strategy Pattern does not is the ability to share inherited methods and attributes. So common code can be shared by putting it in the parent abstract class. These advantages were instrumental in our decision to implement the Template Method Pattern.

Non-functional requirements

Color Grid showing heat simulation of the heat dispersion



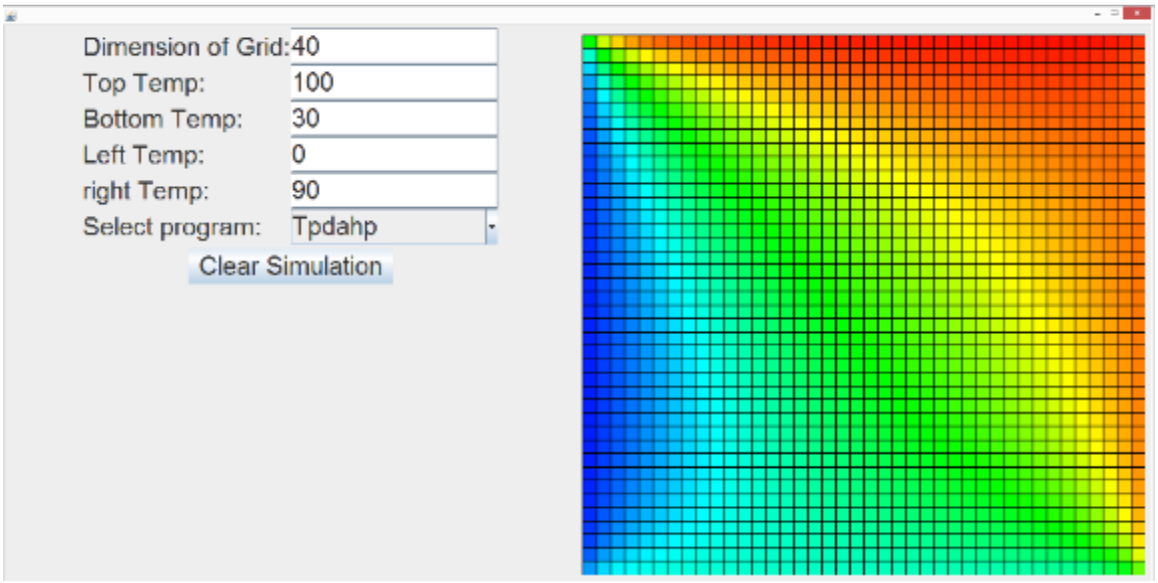
For the non-functional requirement of usability, we chose to implement a GUI that would display the results with a color grid which updates as the algorithm runs. The colors in the grid change as the heat is dispersed throughout the plate. This give's the user a visual understanding of the dispersion of heat.

For the non-functional requirement of evolvability, we chose to implement the Template Method Pattern. This design pattern gave the ability to add new algorithm with no change the the GUI and also the ability for the algorithms to share common code in a parent class.

GUI design

The user interface has simple text field inputs for the size of the grid, for the four starting temperatures and a drop down to select which program to run. We chose these controls because they are simple and standard Swing components which we can also validate data entry. For the visualization we chose to display a square grid made up of cells that match the grid size and display the temperature in each cell as a color. The colors range from blue which represents the value 0 to red which represents the value 100 and the cells reflect the appropriate color variation as the experiment proceeds. We chose this visualization method because colors are often used in the industry to convey heat simulations and therefore would be an appropriate interpretation of heat diffusion on a metal plate.

Figure 1: GUI layout



Reflection

After implementing the Template Method Pattern we realized that the algorithm did not have much code that could be shared in the parent abstract class. We were able to use the abstract parent class to store some of the code that displays the grid in the GUI therefore making use of the Template Method Pattern to some extent. The important part of the design was the factory class, which returns a specific implementation of the algorithm based on the user selected input. This is what allowed the GUI to make use of all the different algorithms without any change to code.