**Cambios sobre el proyecto original no enfocados en nuevas funcionalidades**

Si bien como idea del proyecto no fue hacer refactors del proyecto original, si se tuvieron que realizar algunos por el hecho de orden y de clases con muchas responsabilidades que complicaban el trabajo de agregar nuevas funcionalidades.

**Archivos**

Una de las principales tareas donde se hizo un refactor de los archivos del proyecto original qsim web, fue sobre el orden de los mismos.

|  |  |
| --- | --- |
| Originalmente no había una separación entre vistas, componentes, editor y helpers, quedando todo sobre un mismo directorio. | Luego, se dividio entre las vistas, los componentes, el editor y la computadora Q |
|  |  |

**Responsabilidades**

Desde el proyecto original del simulador, se utiliza una clase “CodeExecutor” que por su nombre se esperaría que tenga la responsabilidad de ejecutar el código del programa escrito.  
En la práctica, esta clase termina teniendo la responsabilidad del editor, del parseo, traducción y ejecución del código como así también el mostrar los resultados y errores post-ejecución.

A lo largo de la cursada, se fue intentando quitar la mayor cantidad de responsabilidades de la clase, teniendo en cuenta las limitaciones de tiempo existentes.

Estado de la computadora Q

Uno de los cambios grandes fue que en un inicio, el estado de la máquina Q post-ejecución se actualizaba dentro de CodeExecutor:

|  |
| --- |
| **CodeExecutor.js** |
| function display\_results() {  setRegisters(computer.state.registers.map(r => {  return {  ...r,  id: `R${r.id}`,  value: hexa(r.value),  details: getDetails(r.value),  updated: updatedRegister(`R${r.id}`, hexa(r.value), registers),  }  }))  setSpecialRegisters([  {  id: "SP",  value: toHexa(computer.state.SP),  details: getDetails(computer.state.SP),  updated: updatedRegister("SP", toHexa(computer.state.SP), specialRegisters),  },  {  id: "PC",  value: toHexa(computer.state.PC),  details: getDetails(computer.state.PC),  },  {  id: "IR",  value: getIR(),  details: [{ key: "IR desensamblado:", value: computer.state.IR\_DESCRIPTIVE }].concat(getDetails(computer.state.IR)),  },  ])  setFlags(  [  { key: "Z", value: computer.state.Z, name: "Zero", updated: updatedFlag("Z", computer.state.Z) },  { key: "N", value: computer.state.N, name: "Negative", updated: updatedFlag("N", computer.state.N) },  { key: "C", value: computer.state.C, name: "Carry", updated: updatedFlag("C", computer.state.C) },  { key: "V", value: computer.state.V, name: "Overflow", updated: updatedFlag("V", computer.state.V) }  ],  )  setMemory(getMemory())  setResult("La ejecución fue exitosa")  } |

Esto se migro a un enfoque donde QComputer sea quien tenga la responsabilidad de actualizar su estado lo cual encapsula la lógica dentro de ella.

|  |
| --- |
| **Qcomputer.js** |
| get\_updated\_flags() {  return [  {  key: "Z",  value: computer.state.Z,  name: "Zero",  updated: this.\_updatedFlag("Z", computer.state.Z),  },  {  key: "N",  value: computer.state.N,  name: "Negative",  updated: this.\_updatedFlag("N", computer.state.N),  },  {  key: "C",  value: computer.state.C,  name: "Carry",  updated: this.\_updatedFlag("C", computer.state.C),  },  {  key: "V",  value: computer.state.V,  name: "Overflow",  updated: this.\_updatedFlag("V", computer.state.V),  },  ];  }  updatedRegister(id, value, registers) {  const register = registers.find((r) => r.id === id);  return Boolean(register) && register.value !== value;  }  get\_updated\_registers() {  return this.state.registers.map((r) => {  return {  ...r,  id: `R${r.id}`,  value: hexa(r.value),  details: getDetails(r.value),  updated: r.updated,  };  });  }  get\_updated\_special\_registers(registers, specialRegisters) {  const isSPUpdated = this.updatedRegister(  "SP",  toHexa(computer.state.SP),  specialRegisters  );  let updatedRegisters = [];  if (isSPUpdated) {  updatedRegisters = registers.map((r) => {  return {  ...r,  updated: isSPUpdated && r.id !== "SP" ? false : r.updated,  };  });  }  return {  specialRegisters: [  {  id: "SP",  value: toHexa(computer.state.SP),  details: getDetails(computer.state.SP),  updated: isSPUpdated,  },  {  id: "PC",  value: toHexa(computer.state.PC),  details: getDetails(computer.state.PC),  },  {  id: "IR",  value: this.getIR(),  details: [  { key: "IR desensamblado:", value: computer.state.IR\_DESCRIPTIVE },  ].concat(getDetails(computer.state.IR)),  },  ],  updatedRegisters,  };  } |

|  |
| --- |
| **CodeExecutor.js** |
| function display\_results(isCycleExecution) {  const sp = setRegistersAndFlags();  setMemory(getMemory());  if (isCycleExecution) {  const irDescriptive =  sp  .find((sp) => sp.id === "IR")  ?.details.find((d) => d.key === "IR desensamblado:")?.value ?? null;  setResult(  "La instrucción:" + irDescriptive + " se ejecutó correctamente"  );  } else {  setResult("La ejecución del programa fue exitosa");  }  }  function setRegistersAndFlags() {  const { specialRegisters: sp, updatedRegisters: r } =  computer.get\_updated\_special\_registers(registers, specialRegisters);  setRegisters(r.length > 0 ? r : computer.get\_updated\_registers());  setSpecialRegisters(sp);  setFlags(computer.get\_updated\_flags());  return sp;  } |

Y esto resulta en que “display\_results” delegue la actualización de estado en una nueva función y a su vez esa función actualice el estado interno en base a lo devuelto por **qcomputer**. Esto resulta en que “display\_results” termine teniendo solo la responsabilidad de ver como mostrar el resultado post-ejecución.

Esto facilito mucho, por ejemplo, en la ejecución detallada de cada instrucción, poder agregar cual instrucción se ejecutó en el resultado de forma sencilla, ya que solamente se busca en los registros especiales (que se obtiene desde “setRegistersAndFlags”) el id correspondiente al “IR” y se muestra.