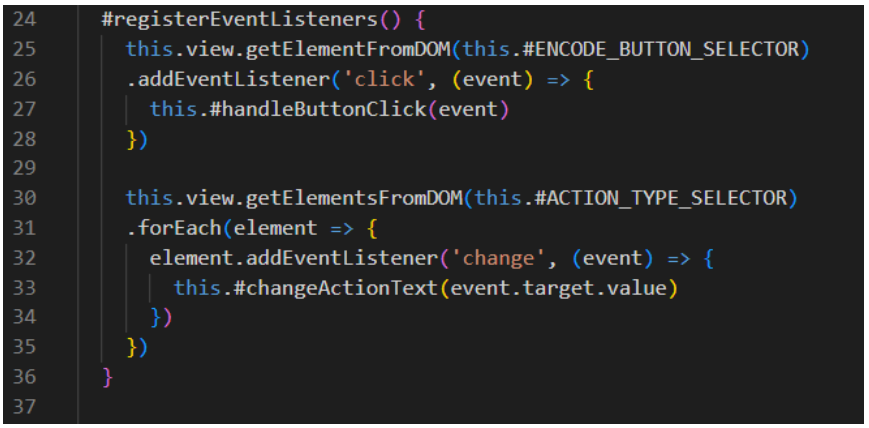
Reflektion kapitel 2 -11 Clean Code

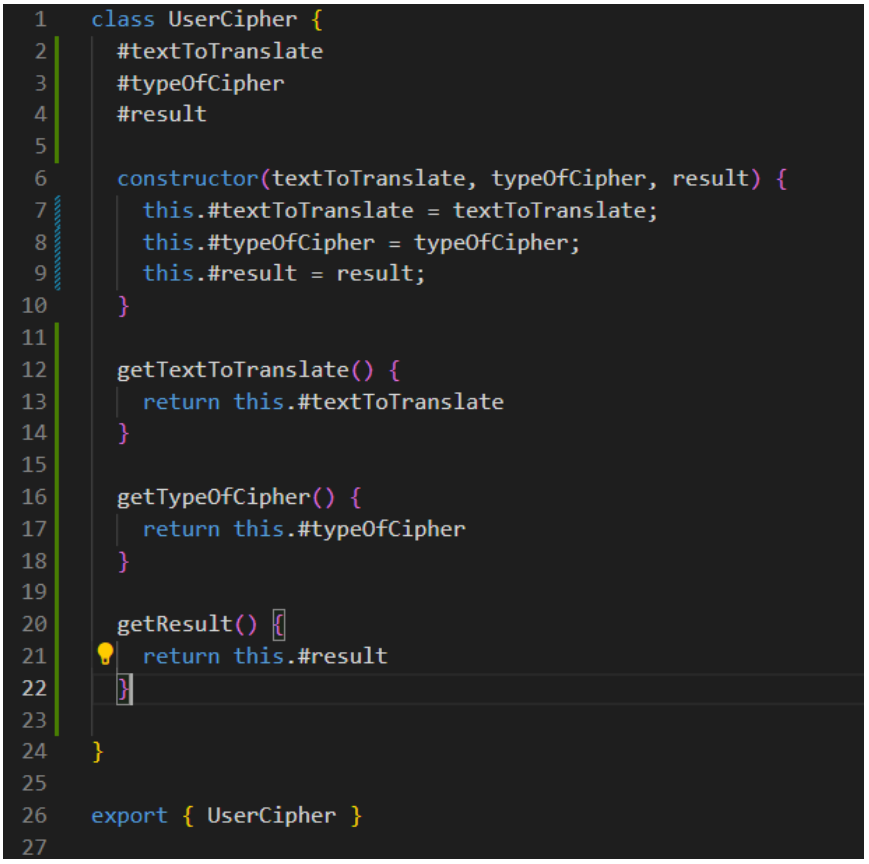
Kapitel	Reflektion
2 - Meaningful names	Detta kapitel var nog det viktigaste för mig. Koden blir så mycket lättare att förstå och läsa när man använder "Intention-Revealing Names". Jag har tidigare haft en tendens att använda "Mental Mapping" (pwd - password, cred - credit). Formatet att tänka Klasser som substantiv och metoder som verb är också väldigt bra. Exempel på "Intention-Revealing Names" och metoder som verb. `#updateHistoryTable() { this.view.renderHistoryTable(this.cipherHistory.getCipherHistory()) } #addCipherToHistory(textToTranslate, typeOfCipher, result) { const userCipher = new UserCipher(textToTranslate, typeOfCipher, result) this.cipherHistory.addCipherToHistory(userCipher) } #clearResultInView() { this.view.updateElementContent(this.#HEADER_RESULT_SELECTOR, '') this.view.updateElementContent(this.#DISPLAY_RESULT_SELECTOR, '') }`
3 - Functions	Här har jag fortfarande utmaningar. Framförallt att en funktion bara ska göra en sak. Att få funktioner små är lättare men att få en abstraktion per funktion kan vara svårare. Jag känner att jag har utvecklat "Top to Bottom" tänket och att använda beskrivande namn faller sig mer och mer naturligt efter kursen. Exempel på att en metod (registerEventListeners) gör en sak samt att använda beskrivande namn. `#registerEventListeners() { this.view.getElementFromDOM(this.#ENCODE_BUTTON_SELECTOR) .addEventListener('click', (event) => { this.#handleButtonClick(event) }) this.view.getElementsFromDOM(this.#ACTION_TYPE_SELECTOR).forEach(element => { element.addEventListener('change', (event) => { this.#changeActionText(event.target.value) }) }) }`
4 - Comments	Boken är ju väldigt radikal vad gäller kommentering av kod och jag kan förstå det ur perspektivet att det är bättre att skapa bra namn på klasser och metoder vilket i de flesta fall gör koden både lättare att läsa och förstå.

	Personligen har jag svårt att läsa t.ex. Java-kod där varje klass och metod är dokumenterad med ett gytter av asterisker och backslashes och där i princip 75 % av kommentarerna är redundanta…
	
	I min app har jag endast kommentar på två ställen och det är när jag sätter ihop en tabell som sen ska renderas i DOM:en:
	
	`#createHistoryTable(cipherHistory) {`
	`  const historyTable = this.createElementInDOM('table')`
	`  const tableHeader = historyTable.createTHead()`
	`  const tableRow = tableHeader.insertRow()`
	
	`  // Create table headers`
	`  this.#TABLE_HEADERS.forEach(header => {`
	`  const tableHeader = document.createElement('th')`
	`  tableHeader.textContent = header`
	`  tableRow.appendChild(tableHeader)`
	`  })`
	
	`  // Create table body`
	`  const tableBody = historyTable.createTBody()`
	`  cipherHistory.forEach(userCipher => {`
	`  const tableRow = tableBody.insertRow()`
	`  const tableData = [userCipher.textToTranslate, userCipher.typeOfCipher, userCipher.result]`
	`  tableData.forEach(data => {`
	`  const tableCell = tableRow.insertCell()`
	`  tableCell.textContent = data`
	`  })`
	`  })`
	
	`  return historyTable`
	`}`
	
	Jag valde att kommentera detta då det är “bakvänd” kod som kan vara svår att förstå. Jag kanske borde refaktorerat “create table headers” och “create table body” till egna metoder, men jag valde att se “tabellen” som ett “objekt” i detta sammanhang.

5 - Formatting	Tack vare de inbyggda formateringsverktygen som finns i t.ex. VSCode behöver jag inte fundera så mycket på formateringen vad gäller t.ex. indrag. Däremot har boken fått mig att fundera på den vertikala “öppenheten” mellan olika koncept. Jag inser att jag är lite ovan vid detta, men när jag väl provar blir det lättare att förstå hur delar i koden hänger ihop. Att använda vertikal “densitet” känns mer naturligt.
	Exempel på indrag och vertikal “öppenhet” och “densitet”:
	
	`1  import { UserCipher } from '../model/UserCipher.js'`
	`2`
	`3  class Controller {`
	`4  #ENCODE_BUTTON_SELECTOR = '#encodeButton'`
	`5  #ACTION_TYPE_SELECTOR = 'input[name="actionType"]'`
	`6  #TYPE_OF_CIPHER_SELECTOR = 'input[name="typeOfCipher"]:checked'`
	`7  #TEXT_TO_CONVERT_SELECTOR = '#textToConvert'`
	`8  #IS_ENCODE_SELECTOR = '#cipher'`
	`9  #TEXT_SPAN_SELECTOR = '#typeOfAction'`
	`10  #HEADER_RESULT_SELECTOR = '#headerResult'`
	`11  #DISPLAY_RESULT_SELECTOR = '#displayResult'`
	`12`
	`13  constructor(cipher, view, cipherHistory) {`
	`14  this.cipher = cipher`
	`15  this.view = view`
	`16  this.cipherHistory = cipherHistory`
	`17`
	`18  this.#registerEventListeners()`
	`19  }`
	`20`

	Vad gäller den horisontella delen brukar inte de verktyg jag använder i VSCode hantera det. Det är lätt att långa “oneliners” blir obegripliga, men genom att radbryta vid “punkterna” mellan metoderna blir det mycket mer lättbegripligt. Exempel på Horisontell “öppenhet” och “densitet”: 
6. Objects and Data Structures	Det blir lätt förvirrat när man pratar om objekt och objekt och det inte är samma sak… Hursomhelst är jag van att använda datastrukturen json för att det inte ska bli felaktiga värden och typer i de indata t.ex. ett API ska använda. I min applikation behöver jag flytta resultatet (textToTranslate, typeofCipher, result) från en användares chiffrering till en tabell med historik. Jag valde då att skapa dels en klass (model) som skapade datan från chiffreringen (UserCipher) och en klass som skapade historiken (CipherHistory). Exempel från klassen UserCipher:

```
24    #registerEventListeners() {
25      this.view.getElementFromDOM(this.#ENCODE_BUTTON_SELECTOR)
26      .addEventListener('click', (event) => {
27        this.#handleButtonClick(event)
28      })
29
30      this.view.getElementsFromDOM(this.#ACTION_TYPE_SELECTOR)
31      .forEach(element => {
32        element.addEventListener('change', (event) => {
33          this.#changeActionText(event.target.value)
34        })
35      })
36    }
37
```

	 Exempel från klassen CipherHistory:  På detta sätt kan jag kapsla in datat och ger dessutom möjlighet att validera det datan som kommer in på ett enklare sätt. Risken för felaktiga värden minskar då.
7 - Error Handling	Som jag skrev i Laboration 2 har jag utmaningar i att använda korrekt felhantering i min kod. Jag valde därför att “omarbeta” min modul så att den istället för att skicka felkoder kastar fel:

Exempel på felhantering i modulen:

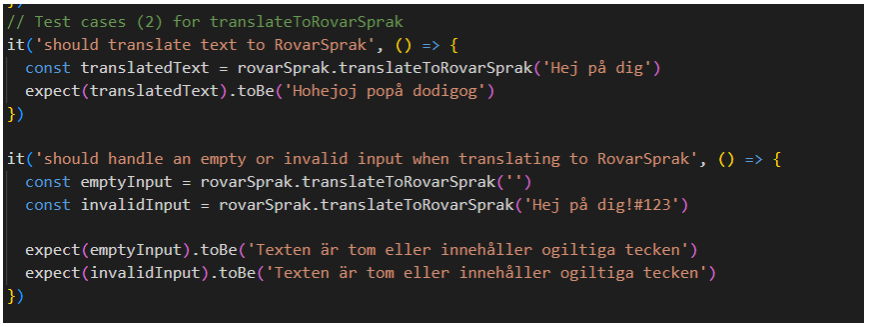
```
class Rovarsprak {
  #charsToSkip = ['a', 'o', 'u', 'å', 'e', 'i', 'y', 'ä', 'ö', '.', ',', '!', '?', ' '

  constructor () {
    this.stringFunctions = new StringFunctions()
  }

  translateToRovarSprak (textToTranslate) {
    let rovarSprak = ''
    if (this.stringFunctions.isStringEmpty(textToTranslate)) {
      throw new Error('Texten är tom')
    } else if (this.validateTextInput(textToTranslate)) {
      for (let i = 0; i < textToTranslate.length; i++) {
        if (this.#charsToSkip.includes(textToTranslate[i])) {
          rovarSprak += textToTranslate[i]
        } else {
          rovarSprak += textToTranslate[i] + 'o' + textToTranslate[i].toLowerCase()
        }
      }
      return rovarSprak
    } else {
      throw new Error('Texten är tom eller innehåller ogiltiga tecken')
    }
  }
```

Felen fångas i min app:

```
if (this.#isEncode()) {
  try {
    result = this.cipher[this.#getCipherToUse()].to(this.#getInputText())
    this.#clearResultInView()
    this.view.clearHistoryTable()

    this.#addCipherToHistory(this.#getInputText(), this.#getCipherToUse(), result)
    this.#updateHistoryTable()

    this.#updateResultInView(result)

  } catch (error) {
    this.#sendFlashMessage(error.message)
  }
```

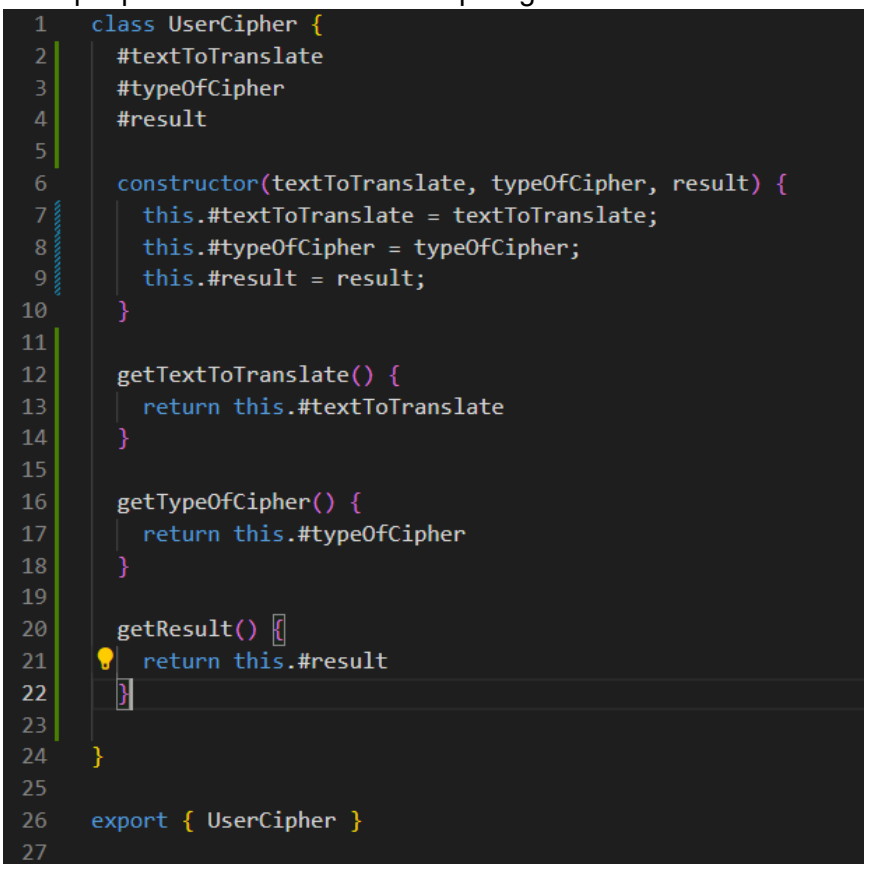
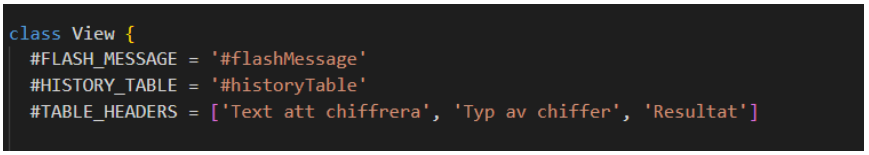
och renderas i vyn som ett “flash”-meddelande:

Här finns utrymme för förbättringar. Jag skulle vilja skapa egna felklasser t.ex. en för varje typ av chiffrering (RovarSprakError, FikonSprakError) för att på det viset kunna hantera/separera felen enklare. Detta skulle både förenkla felsökning och testning.

8 - Boundaries	Det här kapitlet handlar om att tänka på de gränser som behövs när man använder tredjepartskod som t.ex. API:er. För att inte skapa beroenden in i applikationen när tredjepartskod förändras är det oerhört viktigt att kapsla in det. Utifrån min egen kod inser jag att jag borde “kapslat in” modulen på ett bättre sätt och där t.ex. hanterat felen. Som det är nu har jag lagt till felkastningen i modulen, men hanteringen av felen sker i appen. En nyttig lärdom för framtida projekt.
9 - Unit tests	Boken beskriver enhetstester och i mitt projekt gjorde jag olika i modul och app: **Modul** Här använde jag automatiska tester med Jasmin som hade en assert per test och ett koncept per test:  **App** Här valde jag att använda manuell testning som jag presentar i en tabell med testfall, input, förväntad output och testresultat: 
10 - Classes	För mig som kommer från funktionell programmering har den objektorienterade världen varit svår att förstå och jag har upplevt den som onödigt krånglig och tidskrävande. Därför har det varit en stor utmaning för mig att skriva den här appen i en MVC-struktur med objektorienterat fokus. Trots mina utmaningar har jag upptäckt det fina i att jobba med klasser och se till att kapsla in “hemligheter” (även om Javascript inte är ett kompilerat språk) och skapa metoder som öppnar upp de saker man vill öppna upp. Det har varit lite svårt att få in tänket med att klasserna ska vara små och bara ha ett ansvarsområde, men jag känner att jag är på god väg. Eftersom jag kodade en del av appen i HTML blir det lätt “Magiska nummer” men att skapa konstanter inom klasserna hjälpte mig med det och på det viset fick jag inga osynliga beroenden bland alla elementID” som används i appen.

```
// Test cases (2) for translateToRovarSprak
it('should translate text to RovarSprak', () => {
  const translatedText = rovarSprak.translateToRovarSprak('Hej på dig')
  expect(translatedText).toBe('Hohejoj popå dodigog')
})

it('should handle an empty or invalid input when translating to RovarSprak', () => {
  const emptyInput = rovarSprak.translateToRovarSprak('')
  const invalidInput = rovarSprak.translateToRovarSprak('Hej på dig!#123')

  expect(emptyInput).toBe('Texten är tom eller innehåller ogiltiga tecken')
  expect(invalidInput).toBe('Texten är tom eller innehåller ogiltiga tecken')
})
```

Testfall beskrivning	Indata	Förväntat utfall	Testresultat
Ingen text chiffreras	Tomt i fält för text, användare klickar på "Chiffrera text"	Meddelandet "Texten är tom" visas	OK
Text utan vokaler chiffreras till Fikonspråk	"hmpf" anges i fält för text, Användare väljer "Fikonspråk" som chiffer och klickar på "Chiffrera text"	Meddelandet "Texten innehåller inga vokaler" visas	OK

	Exempel på en liten klass med inkapsling.  Exempel på klass med privata konstanter: 
11 - System	Boken diskuterar vikten av att separera delarna i koden eftersom det lätt blir komplext och man klarar inte av att hantera alla detaljer själv. Då min applikation är uppbyggd enligt MVC blir det lätt att använda "Separation of Concerns” Att ha en main-metod visar tydligt hur applikationen “flödar”. I min applikation har jag en app.js som instansierar min Controller som startar applikationen i index.html:

```
class UserCipher {
  #textToTranslate
  #typeOfCipher
  #result

  constructor(textToTranslate, typeOfCipher, result) {
    this.#textToTranslate = textToTranslate;
    this.#typeOfCipher = typeOfCipher;
    this.#result = result;
  }

  getTextToTranslate() {
    return this.#textToTranslate
  }

  getTypeOfCipher() {
    return this.#typeOfCipher
  }

  getResult() {
    return this.#result
  }

}

export { UserCipher }
```

```
class View {
  #FLASH_MESSAGE = '#flashMessage'
  #HISTORY_TABLE = '#historyTable'
  #TABLE_HEADERS = ['Text att chiffrera', 'Typ av chiffer', 'Resultat']
```

```
import { Controller } from './controller/controller.js'
import { Cipher } from '../model/cipher_module.js'
import { View } from '../view/View.js'
import { CipherHistory } from '../model/CipherHistory.js'


const app = new Controller(Cipher, new View(), new CipherHistory())

export { app }
```

I min Controller använder jag “Dependency Injection” för att koppla ihop Controllern med Modellen och Vyn:

```
import { UserCipher } from '../model/UserCipher.js'

class Controller {
  #ENCODE_BUTTON_SELECTOR = '#encodeButton'
  #ACTION_TYPE_SELECTOR = 'input[name="actionType"]'
  #TYPE_OF_CIPHER_SELECTOR = 'input[name="typeOfCipher"]:checked'
  #TEXT_TO_CONVERT_SELECTOR = '#textToConvert'
  #IS_ENCODE_SELECTOR = '#cipher'
  #TEXT_SPAN_SELECTOR = '#typeOfAction'
  #HEADER_RESULT_SELECTOR = '#headerResult'
  #DISPLAY_RESULT_SELECTOR = '#displayResult'

  constructor(cipher, view, cipherHistory) {
    this.cipher = cipher
    this.view = view
    this.cipherHistory = cipherHistory

    this.#registerEventListeners()
  }

```

Detta skapar flexibilitet och underlättar om man t.ex. vill testa nya vyer.