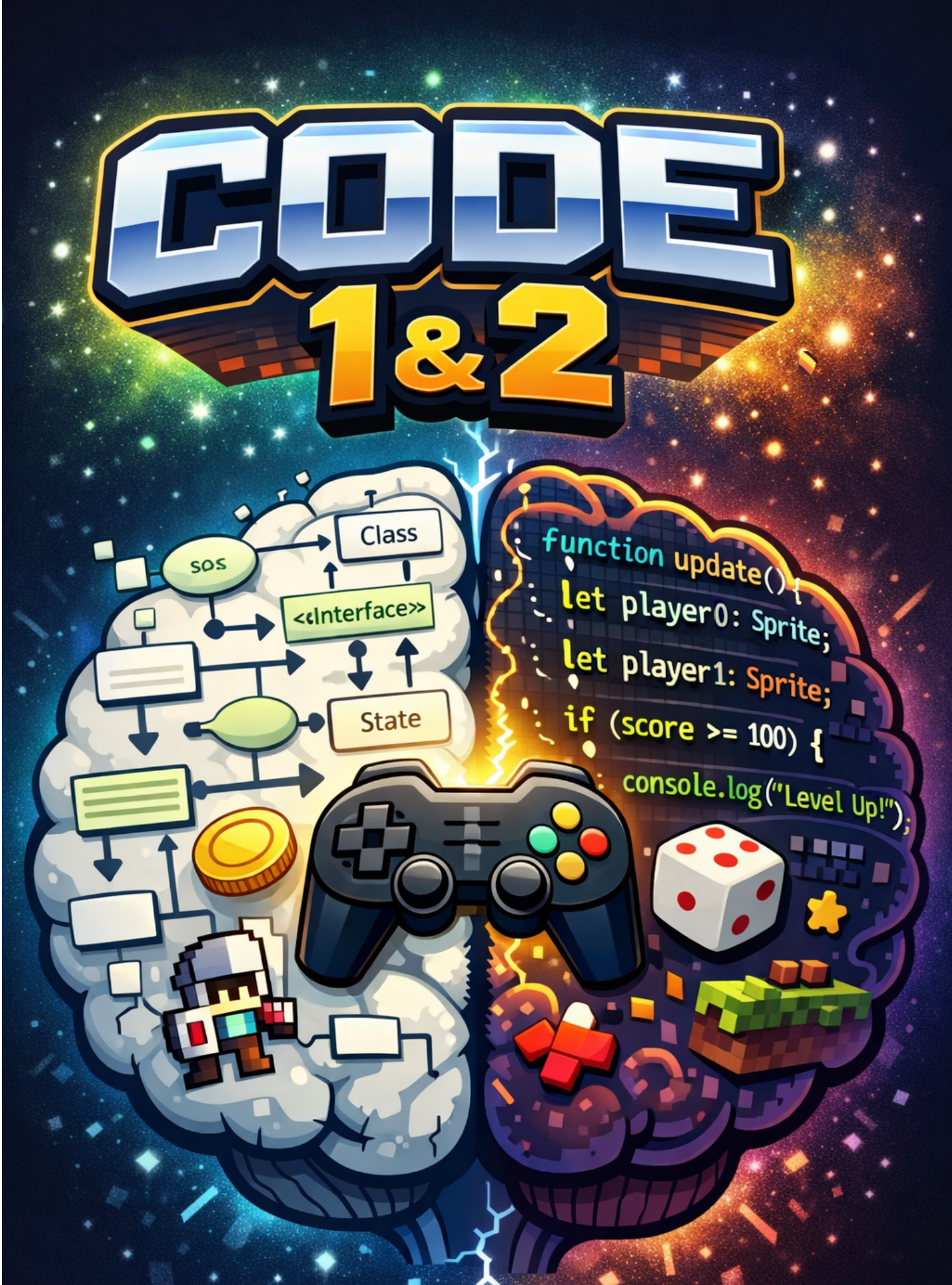
CODE 1&2



Prof. Dipl.-Ing. Jirka Dell'Oro-Friedl | Hochschule Furtwangen | The Black Forest University | V1.0 © 5/2026

Inhaltsverzeichnis

1 Foundation

1.1 Hypertext Markup Language

HTML is not a programming language. It's just a format to hierarchically structure documents and provide basic interaction like hyperlinks. In its most basic form, an HTML document consists of the actual text content and markups to semantically tag it. This way, parts of the text can be marked as headlines or paragraphs or hyperlinks etc.

These markups follow a strict and simple syntax: a tag name in pointy brackets followed by an optional set of attributes, given as key-value pairs. All values written in quotation marks. The content follows, its end marked again with pointy brackets and the same tag name, preceded by a slash. For tags without content, the slash precedes the closing bracket.

```
1   <tag attribute1="value1" attribute2="value2" ...>
2     text content or
3     <anothertag .../>
4   </tag>
```

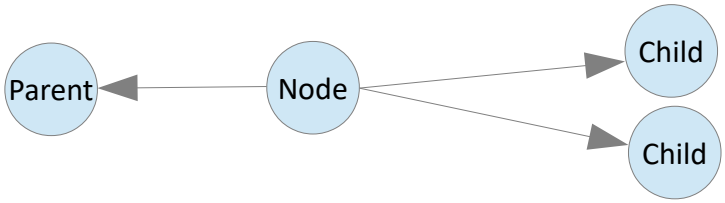
Line 1 contains an opening tag which gets closed in line 4. Thus, it marks the content in the lines 2 and 3. Of course, it may contain many more lines and even other tags as in line 3. Such a nested tag may again contain text and other tags, creating a hierarchical structure. In the example, the tag on line three does not contain anything and therefore ends with a slash and the closing bracket, which spares the extra closing tag. For example, the tag `image` usually refers to an image to load and has no content.

There are numerous predefined tags, some provide complex functionality like form elements or the video tag.

1.2 Document Object Model

When the browser loads an HTML-file, it parses its content and creates an internal representation: the document object model. It then tries to present this model to the user via output devices.

Tags divide the content into so called nodes. Each node is a set of data, including a list of references to the nodes nested within, called child nodes, and a reference to the node that contains it, called parent node.



The example above translates into a similar structure, with lines 2 and 3 creating the two child nodes and the embracing tag creating the node in the center as their parent. That node sure refers to its parent, though it is not shown above.

The minimal graph of the DOM consists of a topmost node with the name “HTML” and two child nodes named “HEAD” and “BODY”. The browser creates this DOM even if it parses a completely empty file. While the body node is is supposed to contain the contents to present, the head node contains additional information and links to other files to parse. More nodes and hierarchical connections between them, called edges, form a tree shaped graph.

Manipulation of this DOM in the computer's memory by a running program provides for a dynamic experience, since the browser renders the changes automatically. The tree's structure may be altered by adding or removing child nodes, or the data of the node may be manipulated, changing content and appearance.

1.3 Cascading Style Sheet

The styles to use to present the content, like fonts and colors, are stored in so called style sheets. The browser's internal default style sheet and parts thereof can be overwritten in various ways. Common and recommended is to keep style information in a separate file and to link this to the HTML file using the link tag in the head section.

```
<html>
  <head>
    <link rel="stylesheet" href="filename.css">
  </head>
  ...
</html>
```

Each style description consists of a set of key-value-pairs separated by semicolons, enclosed by curly brackets which are preceded by a so called selector.

```
selector {
  color: red;
  ...
}
```

In its simplest form, the selector is a tag name. All tags with this name will be styled as described. Selectors can become quite complex, addressing attributes of or the relationship between nodes and more. Selectors may also be used at runtime to find matching nodes in the DOM using the **`querySelector(…)`** command.

A style applied to a node is also applied to its child nodes, as long as it is not overwritten by another style. Multiple style sheets may be used to overwrite previously defined styles. Once parsed, each node of the DOM holds the style used to render it and a running program may change this style at runtime. This cascade of possible style changes is the reason for the name.

1.4 JavaScript

JavaScript was originally designed by Brendan Eich in 1995 – in just ten days – to add some visual effects to web pages. Since then, it became the most popular programming language on GitHub today. It is

standardized	As ECMA-Script since 1997
imperative	An algorithm is implemented as a sequence of explicit steps following one after the other
high level	It is easily human readable and implements intuitive concepts
interpreted	Executed directly from the human readable file, no explicit process to compile a machine readable program first is necessary
dynamically typed	The type of data may change during the execution of the program
implicitly typed	The type of data is not written in the program code but inferred at runtime
object-oriented	Combining procedures and data to complex entities
platform agnostic	Since it is standardized and interpreted, it does not only run in browsers but on various platforms from servers to micro controllers

To run JavaScript in a browser, the preferred method is to simply link the file with the code to the HTML-file using the script-tag in the head section

```
<html>
  <head>
    <script src="filename.js"></script>
  </head>
  ...
</html>
```

The JavaScript file is just a text file with a sequence of statements

```
let greet = "Hello World";
console.log(greet);
```

With dynamic and implicit typing, together with the interpreters tolerance towards programming flaws, JavaScript offers an easy introduction to programming. As programs become more complex, this quickly reverses into a disadvantage, since students do not train proper coding and even basic errors show far too late.

1.5 TypeScript

TypeScript extends JavaScript with modern programming paradigms, strict type checking, intelligent design time error detection and much more. However, a program written in plain JavaScript is also a valid TypeScript program. This way, it is possible to start simple and then gradually add more enhancements to prepare for proper and efficient coding of large and complex applications like games. The development of TypeScript started in 2012, lead by Anders Heilsberg, who is also known for C#. Today, it is the third most popular language on GitHub over all economies.

The modules Code 1 and Code 2 of the study program “Games & Immersive Media” require the use of TypeScript with explicit typing. The use of the type `any` is prohibited with just very few exceptions. Here's the code above written in TypeScript:

```
let greet: string = "Hello World";
console.log(greet);
```

Using explicit typing, in this case declaring the variable `greet` with the annotation of the type `string` (separated from the variable name by a colon), enables the TypeScript compiler to detect and show type errors at design time. It then creates a JavaScript file from valid TypeScript code, which can be linked to an HTML-file as shown above.

1.6 Operators

This is a list of the most important operators in TypeScript / JavaScript

Op	Example	Description	
	1.6.1 Assignment		
=	`x = 7;`	The value on the right is assigned to the variable on the left side.	
	1.6.2 Arithmetic		
+	`x + 5;`	Addition	Yields the sum of the left and right values without storing it

Op	Example	Description	
-	`17 - x;`	Subtraction	Yields the difference of the left and right values without storing it
*	`5 * x;`	Multiplication	Yields the product of the left and right values without storing it
/	`x / 2;`	Division	Yields the quotient being the result of the left value divided by the right value without storing it
%	`x % 10;`	Modulo	Yields the remainder of the division of the left value by the right value without storing it
	1.6.3 Combined		
+=	`x += 19;`	Addition Assignment	The variable's value is increased by the value on the right. Equivalent to `x = x + 19;`
-=	`x -= 3;`	Subtraction Assignment	The variable's value is decreased by the value on the right. Equivalent to `x = x - 3;`
*=	`x *= 100;`	Multiplication Assignment	The variable's value is multiplied by the factor on the right. Equivalent to `x = x * 100;`
/=	`x /= 4;`	Division Assignment	The variable's value is divided by the divisor on the right. Equivalent to `x = x / 4;`
++	`x++;`	Increment	The variable's value is increased by 1. Equivalent to `x += 1;`
--	`x--;`	Decrement	The variable's value is decreased by 1. Equivalent to `x -= 1;`
	1.6.4 Comparison		
==	`(x == "AB")`	Value Equality	Returns true if the values on the left and right side are equal. Caution with floats!
===	`(x === "4")`	Strict Equality	Returns true if the values are equal and both expressions are of the same type.
!=	`(x != "AB")`	Inequality	Returns true if the values on the left and right side are different.
>	`(x > 2.32)`	Greater Than	Returns true if the left value is greater than the right.
<	`(x < 2.32)`	Less Than	Returns true if the left value is less than the right.

Op	Example	Description	
>=	(x >= 2.32)	Greater Than or Equal To	Returns true if the left value is greater than or exactly equal to the right.
<=	(x <= 2.32)	Less Than or Equal To	Returns true if the left value is less than or exactly equal to the right.
	1.6.5 Logic		
&&	x>2 && x<9	And	Returns true if both the left and the right expressions are true. Here, if the value of x is between 2 and 9
||	x<2 || x>9	Or	Returns true if at least one of the two expressions is true. Here, if the value of x is outside the range 2 to 9
!	! (x > 10)	Not	Negates the expression, meaning it returns true if the following expression is false. Here, if x <= 10

2 Unified Modeling Language

Game Designers and Developers need a precise and international language to create ideas and convey their designs to a team of developers. The Unified Modeling Language (UML) fulfills this requirement not only in the realm of software development, but for arbitrary complex systems. Its development started in the 1990s and is still going on. It is standardized as ISO/IEC 19505. UML makes systems, algorithms and data structures visible and tangible, thus enabling the designers and their teams to discuss, improve and produce them not only in early stages of development, but throughout the process, and, done well, even produces in large parts the final documentation.

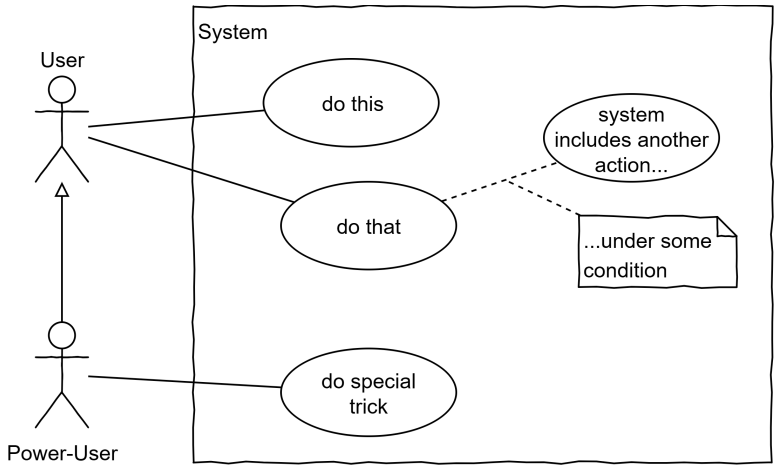
This little booklet displays only three of the many types of diagrams UML 2.5 defines:

- Use Case Diagram
- Activity Diagram
- Class Diagram

and of those only a subset of the features. This resembles the minimum expertise a designer for digital media needs to master and should be sufficient to design systems of non trivial complexity.

2.1 Use Case Diagram

A use case diagram models a system on a very high level of abstraction from the perspective of the user. The system is modeled as a black box, offering interactions to the user. There may be various user roles using different interactions the system offers.



User roles may be extended to access more interactions, e.g. Power-User extends User. Use cases may be internal (not started by users), which can also be depicted in the diagram and connected by a dashed line. Notes can further specify conditions or additional information. Make sure not to go too much into details in the use case diagram. Stay on a level that provides a good overview!

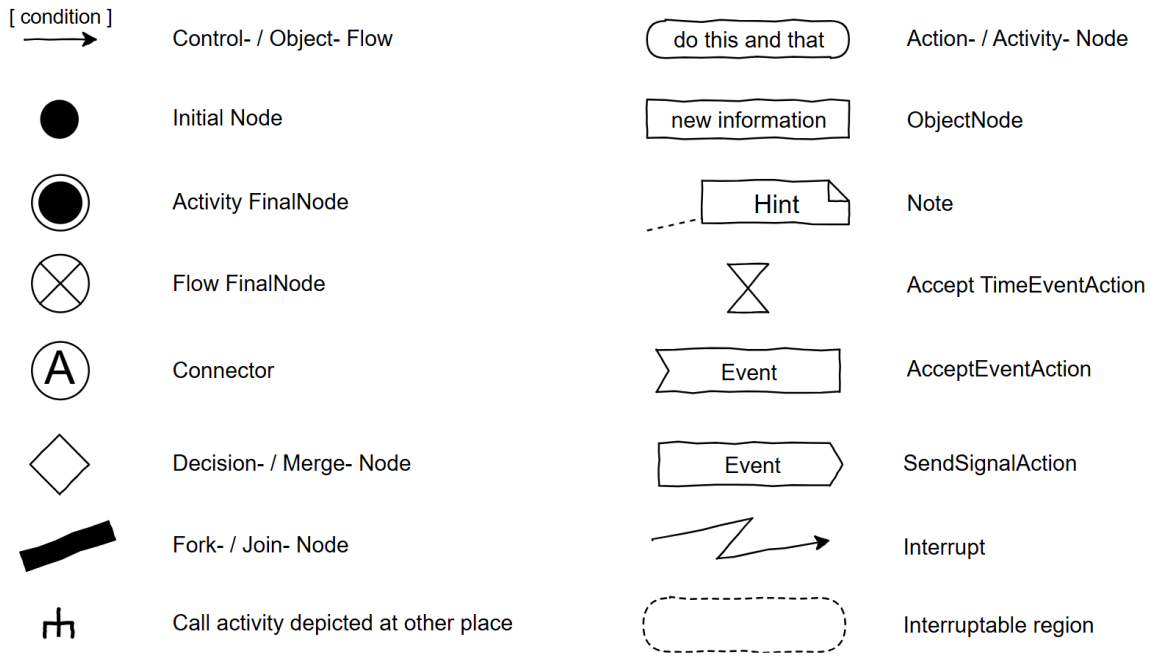
2.2 Activity Diagram

This type of diagram is arguably the most versatile of the UML diagrams. An activity diagram models the behavior of a system. While natural language is only suited to describe simple processes, using the graphical language in two dimensions and nesting (sub-activities) enables the designer to describe arbitrary complexity while maintaining readability and comprehensibility. In this chapter, designs displayed are on a very low level, so that they can

be translated into very low level code. This is for explanation only and in real life, the designer will not go down into that much detail. The same structures apply though when working with activities that nest atomic actions. By nesting activities in again larger activities and so forth, the designer works on different levels of abstraction.

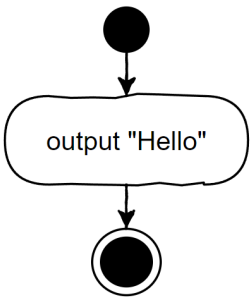
However, in the design process, the designer starts with the activities defined in the use case diagram, splits them up in smaller ones, and those again in even smaller, until all activities designed are trivial and the initial problem is solved.

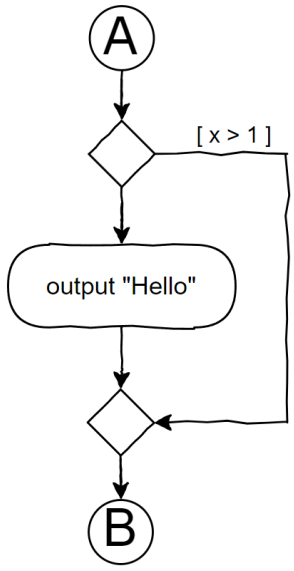
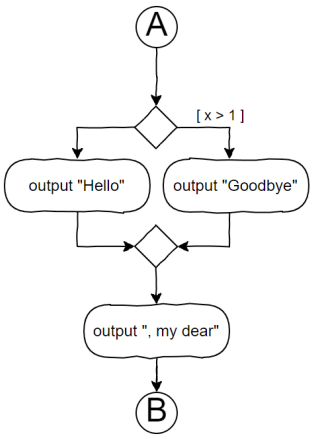
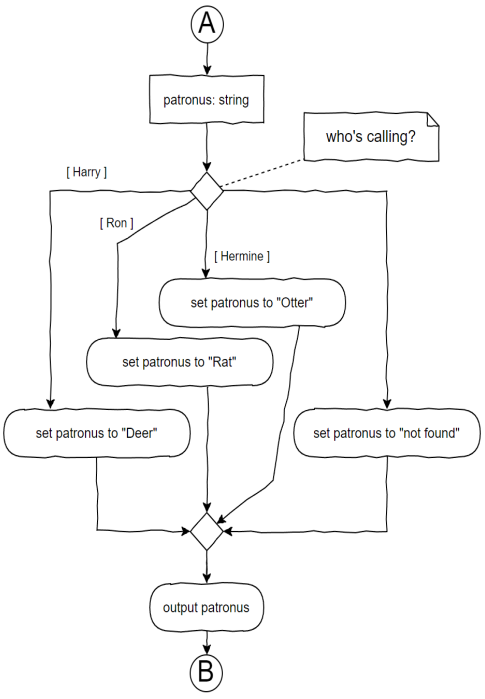
2.2.1 Elements



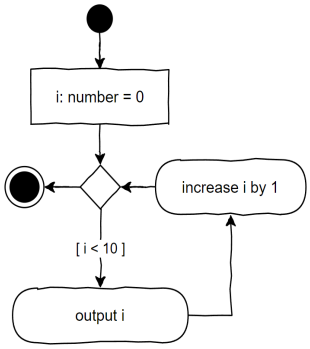
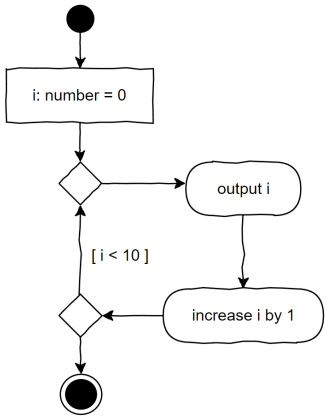
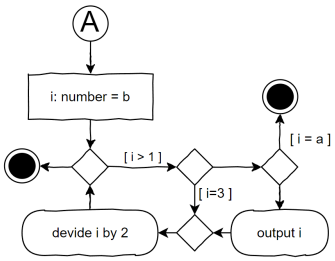
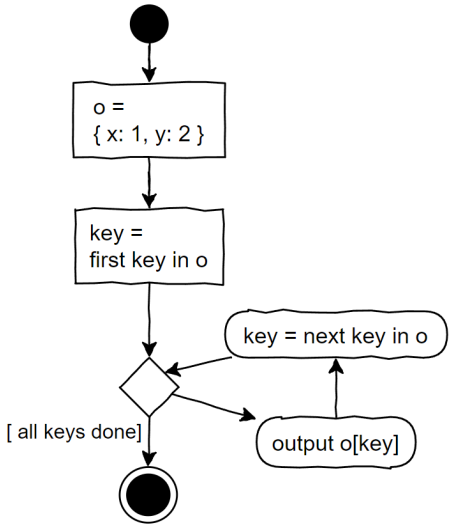
2.2.2 Examples diagrams with TypeScript code

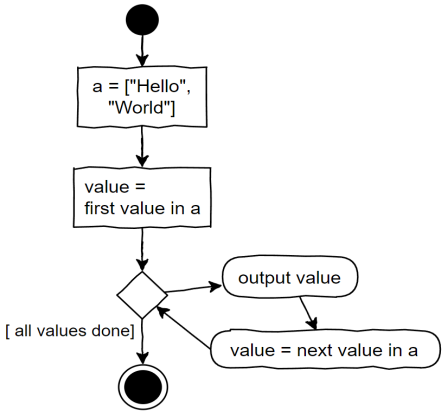
2.2.2.1 Conditions

Type	Diagram	Code
Un-conditional linear flow		`console.log("Hello");`

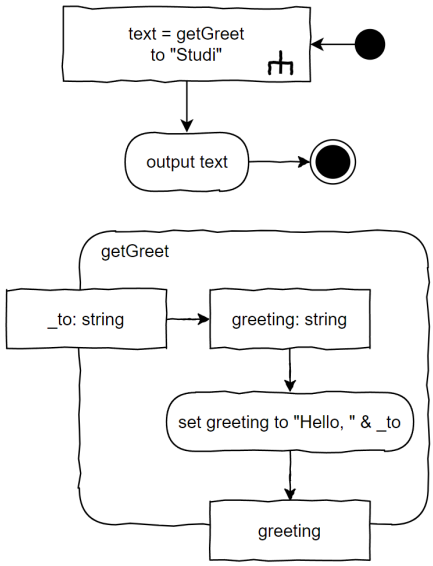
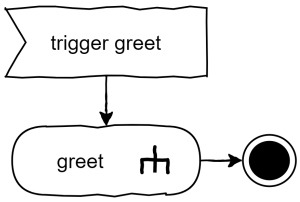
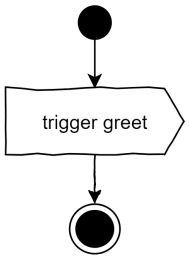
Type	Diagram	Code
Conditional		``` ... if (!(x > 1)) console.log("Hello"); ... ```
Exclusive Conditional		``` ... if (x > 1) console.log("Goodbye"); else console.log("Hello"); console.log(", my dear"); ... ```
Multiple Conditions		``` ... let patronus: string; switch (person) { case "Harry": patronus = "Deer"; break; case "Hermine": patronus = "Otter"; break; case "Ron": patronus = "Rat"; break; default: patronus = "not found"; break; } console.log(patronus); ... ```

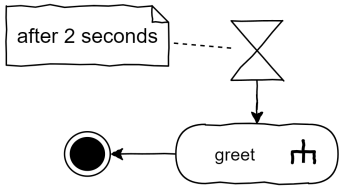
2.2.2.2 Loops

Type	Diagram	Code
Pre Test		`let i: number = 0;` `while (i < 10) {` `  console.log(i);` `  i++;` `}` or `for (let i: number = 0; i < 10; i++)` `  console.log(i);`
Post Test		`let i: number = 0;` `do {` `  console.log(i);` `  i++;` `} while (i < 10);`
Complex Control		`...` `for (let i: number = b; i > 1; i/=2)` `{` `  if (i == 3)` `  continue;` `  if (i == a)` `  break;` `  console.log(i);` `}`
Iteration over all keys of an associative array or all indices of a simple array		`let o = { x: 1, y: 2};` `for (let key in o) {` `  console.log(o[key]);` `}` Note: The for..in loop works also with simple arrays, yielding the index as key

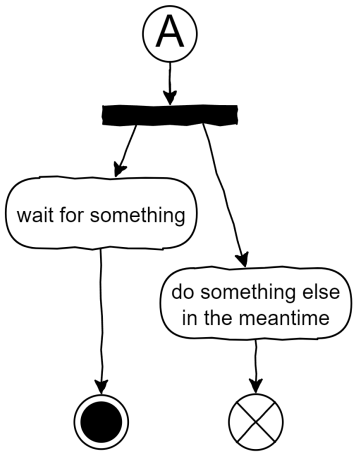
Type	Diagram	Code
Iteration over all values of a simple array		`let a = ["Hello", "World"];` `for (let value of a) {` `  console.log(value);` `}`

2.2.2.3 Sub-activities and Signals

Type	Diagram	Code
Call a subactivity with input and output		`let text: string = getGreet("Studi");` `console.log(text);` `function getGreet(_to: string): string {` `  let greeting: string;` `  greeting = "Hello, "+_to;` `  return greeting;` `}`
Accept Event		`someEventTarget.addEventListener(` `  "triggerGreet", greet` `);`
Send Signal		`let event: Event =` `  new Event("triggerGreet");` `someEventTarget.dispatchEvent(event);`

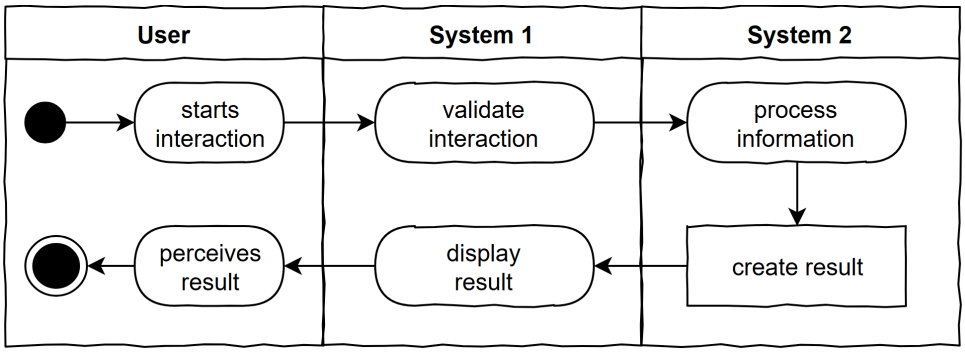
Type	Diagram	Code
ccept Time Event		`window.setTimeout(greet, 2000);`

2.2.2.4 Parallel Processing

Diagram	Code
	`waitForSomething();` `doSomethingElse();` `async function waitForSomething():` `Promise<void> {` `  console.log("starting to wait for something")` `  await something();` `  console.log("done waiting for something");` `}` `function doSomethingElse(): void {` `  console.log("doing something else")` `}`

2.2.3 Activity Partitions

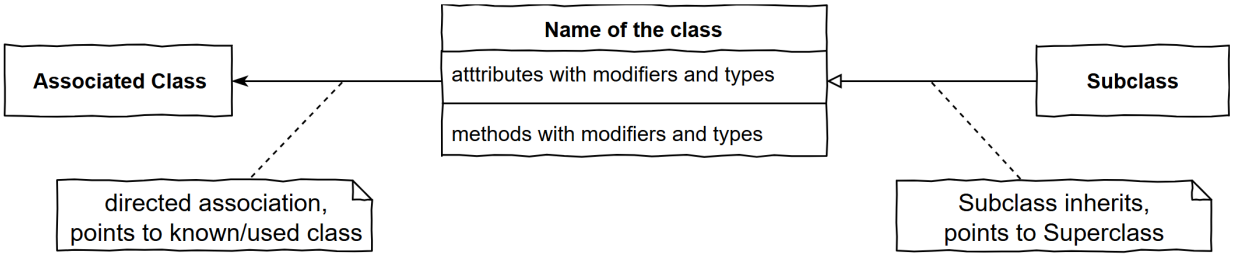
Activity diagrams may stretch over different domains, for example when several systems communicate and exchange data. Write the domain names as the headers of so called “swim lanes” and draw the activities and actions in those lanes accordingly. A user may also be a domain in such a partitioned diagram.



2.3 Class Diagram

The class diagram models complex structures of information used in the designed system. The focus is less on the activities within the system, but on the data those activities are applied to. Designing these structures well is crucial for the performance, stability, maintainability as well as the feasibility of the system. Thus, in complex systems, the class diagram and the activity diagrams are created in parallel, each reflecting on the other.

2.3.1 Structure



Carefully consider the arrows, they symbolize different relationships. The right arrow symbolizes inheritance and points to the generalization. The left arrow is some kind of association, in this case objects of the class in the center know, use or contain objects of the associated class. This can also be bidirectional or, if the relationship is not yet to be further specified, just a line without a direction.

2.3.2 Design

The designer tries to identify classes of objects in the domain of the problem to be solved. During this process it is required to iterate over these five questions over and over again. Considering one object …

Question	**Infers**
what does it have?	attributes, properties
what can it do?	methods
what does it know?	parameters to methods, entanglement
what is it?	inheritance, polymorphism
who cares?	owner, maintainer, creator

When answering those questions, the designer must strive to make every part of the system…

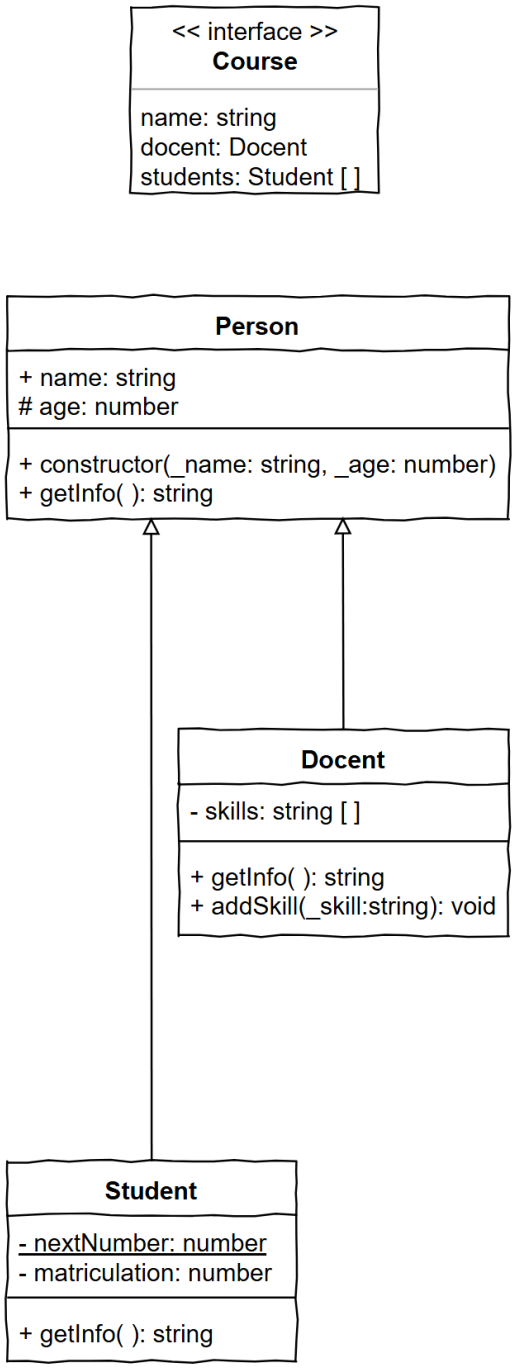
- only as “smart” as necessary
- as “dumb” as possible

While doing so the designer makes use of the four principles of object-oriented modeling:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

2.3.3 Example diagram with TypeScript code

Class Diagram



Code

```
interface Course {
  name: string;
  docent?: Docent;
  students: Student[];
}

class Person {
  public name: string;
  protected age: number;
  public constructor(
         _name: string, _age: number) {
    this.name = _name;
    this.age = _age;
  }

  public getInfo(): string {
    return this.name;
  }
}

class Docent extends Person {
  private skills: string[] = [];
  public getInfo(): string {
    return "Prof. " + super.getInfo()
           + ", age: " + this.age;
  }

  public addSkill(_skill: string): void {
    this.skills.push(_skill);
  }
}

class Student extends Person {
  private static nextNumber: number = 0;
  private matriculation: number;
  public constructor(_name: string,
                     _age: number) {
    super(_name, _age);
    this.matriculation =
Student.nextNumber;
    Student.nextNumber++;
  }

  public getInfo(): string {
    return  this.matriculation + ": "
            + super.getInfo();
  }
}
```

Example Main Program

```
let courses: Course[] = [];
let course: Course = { name: "Physics", students: [] };
course.docent = new Docent("Einstein", 71);
course.docent.addSkill("Relativity");

let student: Student = new Student("Heisenberg", 49);
course.students.push(new Student("Hawking", 8), student);
courses.push(course);
courses.push({
  name: "Art",
  students: [student, new Student("Dali", 46)]
});

for (let course of courses) {
  console.log("Course: " + course.name);

  if (course.docent)
    console.log("• Docent: " + course.docent.getInfo());
  else
    console.warn("• No docent assigned to this course");

  for (let student of course.students)
    console.log("• Student " + student.getInfo());
}
```

Output

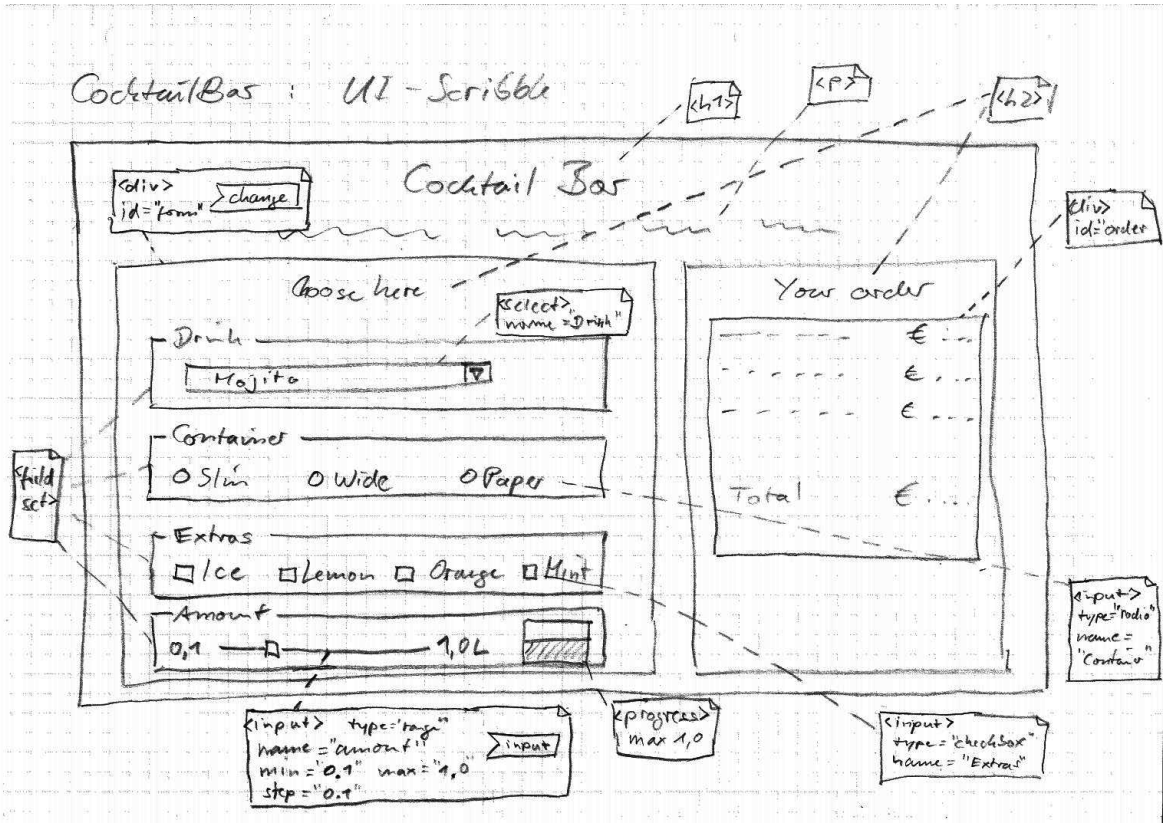
```
Course: Physics
• Docent: Prof. Einstein, age: 71
• Student 1: Hawking
• Student 0: Heisenberg
Course: Art
• No docent assigned to this course
• Student 0: Heisenberg
• Student 2: Dali
```

2.3.4 Modifiers

Symbol/Format	Meaning	Description
+	public	accessible from anywhere
#	protected	objects of the class or its subclasses have access
-	private	only objects of the class have access
underlined	static	belongs to the class, not to an object of the class
italic	abstract	concrete functionality needs to be defined in subclass
<< interface >>	specifies interface	describes a data structure without actual data
<< enum >>	specifies enumeration	describes a specific set of values for the data type

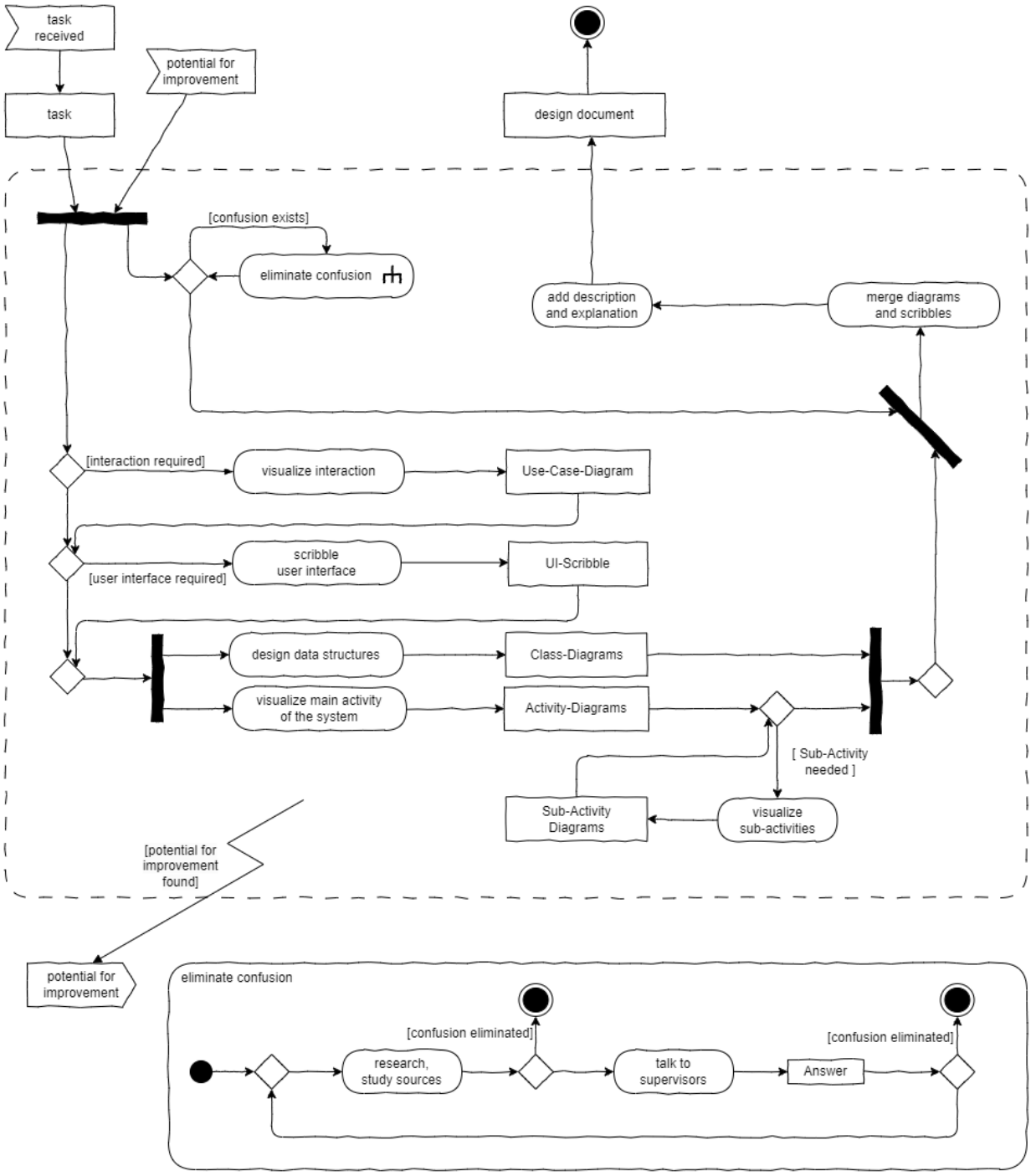
3 UI-Scribble

In order to design interactive applications there is at least one more thing necessary, which is not part of the UML-specification. At least for applications with a visible user interface, this needs to be defined and is done so with UI-Scribbles.
UI-Scribbles depict the layouts of screens and point out the elements used, especially those for interaction. It's far less about colors and shapes than about defining the core structures the following design process builds on. That's why this step comes right after the Use-Case-Diagram.



For a user interface to be created with standard DOM-elements, annotations in the scribble hint to the types of elements to use and possibly additional information like identifiers, css-classes and other attributes. Most importantly, the annotations define which elements will receive signals about the user interaction. Thereby, the starting points of the algorithms processing the user interaction are also defined. See the symbols for accepting the signals "change" and "input" in the example above.

4 Design Process



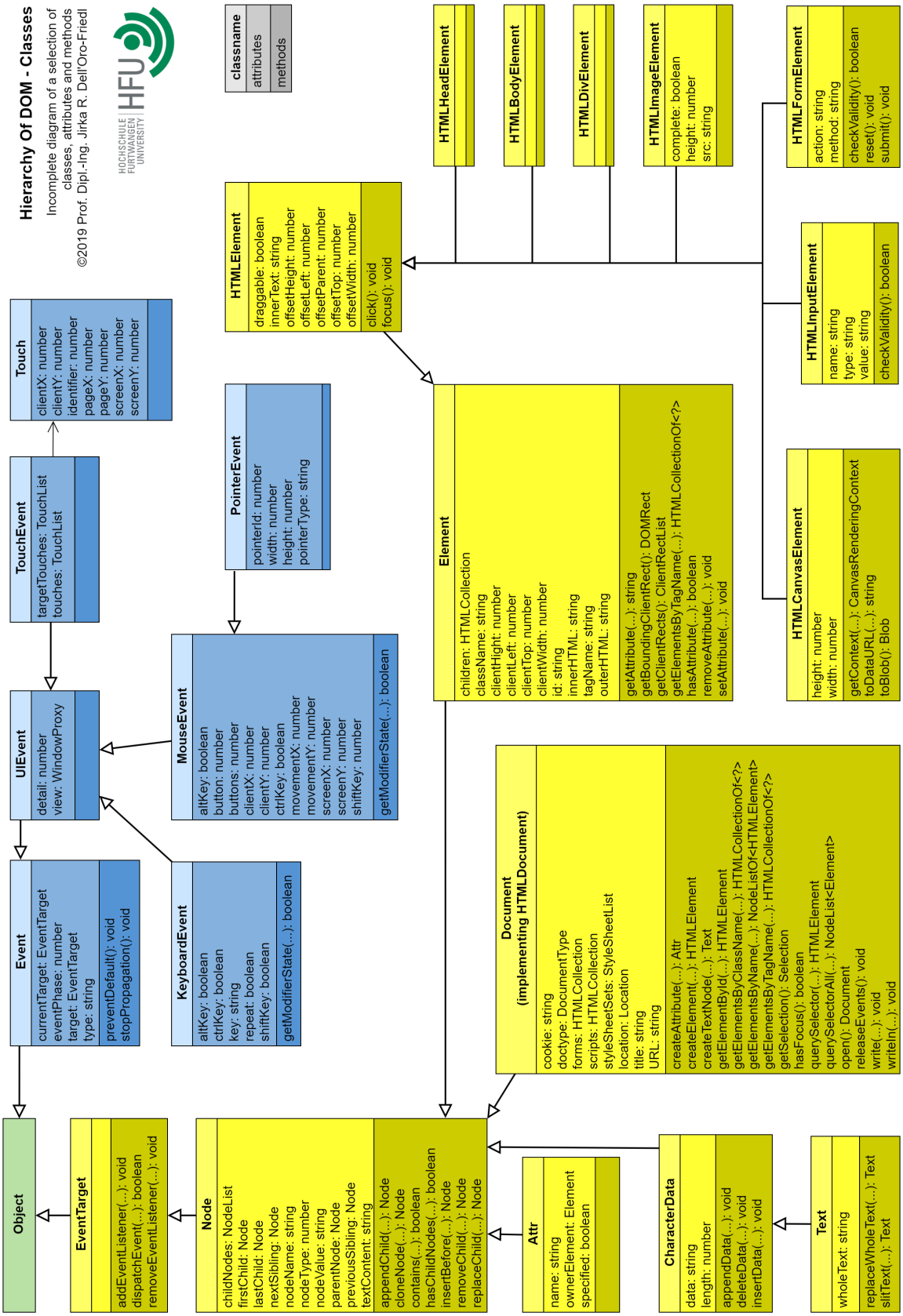
5 Style

<table>
<tr><th>Marker</th><th>Explanation</th></tr>
<tr><td></td><td><h3>5.1 Naming</h3></td></tr>
<tr><td>Self-explanatory</td><td>The code helps a reader to understand it. This requires the strict use of naming conventions. Use names that clearly explain the activity or information addressed and don’t be stingy with letters. Short names are allowed only in very small scopes or when their meaning is clear by convention, such as <code>y</code> for a vertical position.</td></tr>
<tr><td>Variables and Functions</td><td>Names start lowercase and follow the camelCase notation, with uppercase letters indicating the start of a new part in a compound name such as <code>animalLion</code>. The names of variables describe an information or an object. Names of functions and methods strictly describe activities e.g. <code>calculateHorizontalPosition(...)</code> or questions e.g. <code>isHit()</code></td></tr>
<tr><td>Formal Parameters</td><td>Name formal parameters in a functions signature like variables, but prefix them with an underscore like <code>_event: Event</code></td></tr>
<tr><td>Classes, Interfaces, Namespaces, Modules</td><td>Names of classes, interfaces, namespaces or modules start with an uppercase letter and then follow the camelCase notation (PascalCase). The name describes exactly one object of that type, not an activity.
E.g. <code>ObjectManager</code>.</td></tr>
<tr><td>Enumeration</td><td>The names of enumerations and their elements are written all uppercase, with underscores separating parts of the name e.g. <code>EVENT_TYPES.EXIT_FRAME</code></td></tr>
<tr><td>Ambiguities</td><td>Bad example from the DOM-API: <code>getElementById(...)</code> vs <code>getElementsByTagName(...)</code>. Only the little s in the middle indicates that one returns a collection, not a single element. Better: <code>getElementCollectionByTagName(...).</code> However, in <code>getElements(...)</code>, the s is clearly visible, since it’s the last letter.</td></tr>
<tr><td>Prefixes</td><td>Some prefixes may be helpful for finding names for variables, use is encouraged
<table><tr><th>Prefix</th><th>Example</th><th>Meaning</th></tr><tr><td><code>n</code></td><td><code>nObjects</code></td><td>an amount</td></tr><tr><td><code>i</code></td><td><code>iObject</code></td><td>an index</td></tr><tr><td><code>x, y, z</code></td><td><code>xPos</code></td><td>a direction or dimension</td></tr><tr><td><code>min, max</code></td><td><code>maxHeight</code></td><td>boundaries</td></tr></table></td></tr>
<tr><td>Context and Redundancy</td><td>For example, state may have different meaning depending on the context. <code>Machine.state</code> indicates something different than <code>Address.state</code>. However, it is redundant to write <code>Machine.stateTheMachineOperatesIn</code> or <code>Address.stateAsThePoliticalEntity</code>, since the context is provided already by the namespaces. Use this instead of implementing redundancies.</td></tr>
</table>

Marker	Explanation
	### 5.2 Structure
Comments	Use comments sparsely! If you feel that some code needs commenting rethink it and the naming of its components. Remember that you need to maintain comments just as you need to maintain code. Otherwise comments are not only useless but obstructive. Comments are allowed in the following cases 1. when using a documentation generator such as TypeDoc 2. to display an explanation in the development environment e.g. when hovering with the mouse 3. to mark regions in the code (use `"#region"` and `"#endregion"`)
Size	A function should not consist of more than 20 lines of code. If possible, split it up into smaller functions each of which has an explanatory name. This way, the calling function consists of multiple calls that are easy to read and interpret, and the concerns are distributed to smaller functions with the same qualities. Also, watch out for the size of classes, beware of monsters! Keep the number of attributes low.
Separation of Concerns	One function/method should care only about one concern and do this well.
Indentation Depth	A function should not indent more than two levels. Use return statements not only at the end, but so called “early outs” and throw exceptions to keep indentation level low.
Top Down	Order functions and methods in such a way, that the call sits above the called function in code. Reading from top to bottom, the code displays the hierarchy of calls making it possible to understand the overall structure first before diving into the details.
Explicit types	Strictly use explicit typing wherever possible. The type `any` is only allowed as an exception in rare cases.
Semicolons	Always end statements with semicolon.
Literal strings	Literal strings should be enclosed in double quotation marks e.g. `"Hello World!"`
Magic numbers	Are simply disallowed. Never use a literal value in a function call when its meaning is not extremely obvious (e.g. `Math.pow(x, 5)` to retrieve `x` to the power of 5). In all other cases, define a variable with an explanatory name and assign the literal value to it. This way, there is a value and a meaning to it, and its value can be changed in a single place.
Files	Use one file per class. Interfaces may be compiled with classes using them and exceptions are allowed for small classes only used within the scope of the file. Use PascalCase for filenames, exactly the same name as the classes.
DRY	Don't repeat yourself. When you find the same code twice in your program, refactor and create a loop or a function etc. with it.

https://github.com/basarat/typescript-book/blob/master/docs/styleguide/styleguide.md
https://github.com/Platypi/style_typescript
https://github.com/excelmicro/typescript

6 Hierarchy of DOM-Classes



7 FUDGE

FUDGE is a lightweight open-source game engine and editor created for educating students in an academic environment in the field of design and development of games and highly interactive applications. It relies solely on web technologies and is entirely written with TypeScript. Therefore applications built with FUDGE should run on any platform with a browser supporting WebGL2 and WebAudio. The editor requires Node.js on the platform to run.

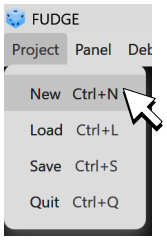
Students with basic knowledge of HTML, CSS, TypeScript and the DOM can easily delve into FUDGE, since it continues the underlying concepts. On the other side, students get acquainted to the additional principles of modern game engines like the entity component system, animation, collision shapes, physics simulation, animation, shaders, component scripts etc.

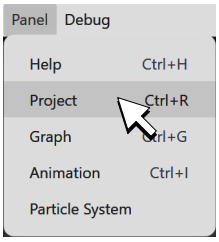
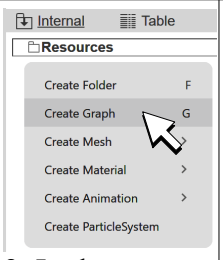
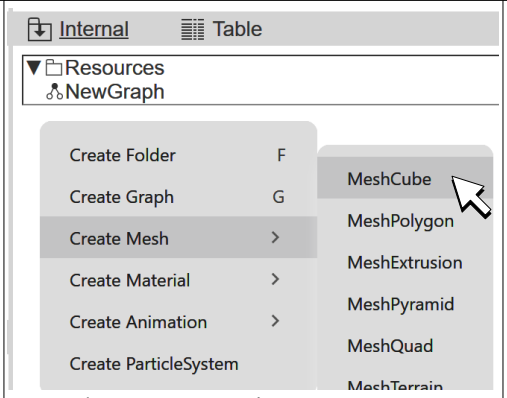
FUDGE is not developed for ease of use, stunning performance or visual fidelity. But for clarity, accessibility, ease of teaching, brevity and quick and easy testing to support fast feedback.

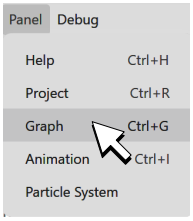
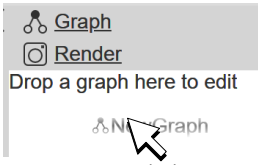
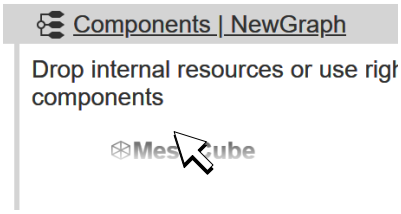
To get an overview and a serious insight into the concepts, browse the FUDGE wiki

7.1 Getting Started Guide

While a smart mix of working with the editor and typing code is required for sophisticated applications, it is possible to use a pure editor based or code based approach to create a minimal application with FUDGE.

	Editor	Code
Setup	 1. Start a new project. A basic setup of files gets created in a new folder.	1. Create an HTML-File 2. Add a canvas tag to the body 3. Link the script https://hs-furtwangen.github.io/FUDGE/Distribution/FudgeCore.js to become independent of future changes, create a local copy and link that. 4. Download the TypeScript definition file to your folder https://fudge.education/Distribution/FudgeCore.d.ts 5. Using TypeScript, create a script and compile 6. Link the JavaScript file created to your HTML file 7. In TypeScript, import FudgeCore e.g. `import ƒ = FudgeCore;` 8. Create a handler to be called after loading the HTML file

	Editor	Code
Create a structure to render meshes	 1. Open the panel "Project"  2. In the resource-view, create a new Graph 3. Save the project. Review the project settings and make sure that graphAutoView is set to your Graph	1. Inside the handler, create an instance of Node with an arbitrary name of your choice 2. Create an instance of ComponentCamera 3. Create an instance of Viewport 4. Call `initialize` on the viewport with an arbitrary name, your node, your camera and a reference to the canvas in the DOM. 5. Call `draw` on the viewport
Test	Load the file index.html to your browser from a local server. You should see a black window and a model of a coordinate system	Load your HTML-file to your browser from a local server. You should see the canvas in black.
Create a mesh and a material to render	 1. In the resource view, create a MeshCube 2. Similarly, create a material. Choose ShaderLit	1. Create an instance of MeshCube with an arbitrary name 2. Create an instance of Material with an arbitrary name and ShaderLit. Omit the Coat, it will then be created automatically When ShaderLit is used, there is no need for extra light sources in the scene. The material will not reflect light but is simply rendered with the color given.

	Editor	Code
Add mesh and material to the scene via compo-nents	 1. Open the panel "Graph"  2. Drag and drop your graph into the render view  3. Drag and drop your resources of the type mesh and material to the component view	1. Create an instances of ComponentMesh with your mesh as the first parameter. Omit the others 2. Create an instance of ComponentMaterial with your material as the parameter 3. Call `addComponent` on your node with each of your components 4. Add another call to `draw` on the viewport
Test	Save the project again and update the browser's output. You should see a white cube now.	Update your browser's output. You should still see a black canvas. The camera can't see the cube since it sits inside of it.
Next steps	You can move the camera interactively using the mouse and the keyboard. FUDGE has setup the environment for you and implemented some functionality. However, this is only the start and most likely not what you want for your game in the end. Now it's time to get into coding. Find the script Main.ts in Script/Source and activate the deactivated lines of code. You should then study the script Autoview.js and transfer to your code what you need and like. In the final version of your application, Autoview.js should not exist any more...	1. Move the camera forward by 5 units calling `translateZ` with a parameter of literal 5 on the pivot matrix of the camera. 2. Now 5 units closer to the viewer, the camera needs to turn around and look back into the scene. Do a 180 degree turn calling `rotateY` on the pivot matrix of the camera. 3. Add another call to `draw` on the viewport 4. Test again. The cube should show now as a white square in the middle of the canvas. You now have coding at your fingertips already. Go ahead fiddling with it. Use ComponentTransform to move whole nodes in the scene, not just their attached components. Add an event listener to the static class Loop and call `start` on it, to get a heartbeat...

8 Resources

- Object Management Group (2015, May): About the Unified Modeling Language Specification Version 2.5. https://www.omg.org/spec/UML/2.5, visited 01/18/2020,
- Kecher, C., Salvanos, A. & Hoffmann-Elbern, R. (2017): UML 2.5: Das umfassende Handbuch. Ausgabe 2018. Inkl. DIN A2-Poster mit allen Diagrammtypen. Bonn: Rheinwerk Verlag GmbH.
- Oestereich, B., & Scheithauer, A. (2013): Analyse Und Design Mit Der Uml 2.5: Objektorientierte Softwareentwicklung (German Edition) (11th 11., Umfassend überarbeitete und aktualisierte Auflage ed.). München: Walter de Gruyter.
- Microsoft. (2020). TypeScript - JavaScript that scales. http://www.typescriptlang.org/, visited 01/18/2020
- Dell'Oro-Friedl, J. et al. (2026): FudgeCore Reference. https://fudge.education/Documentation/Reference/Core/, visited 03/31/2026
- Dell'Oro-Friedl, J. et al. (2026): Fudge Wiki. https://github.com/hs-furtwangen/FUDGE/wiki, visited 03/31/2026
- Dell'Oro-Friedl, J. et al. (2026): Fudge. https://fudge.education/, visited 03/31/2026