Principles of Software Construction: Objects, Design, and Concurrency

Basic GUI concepts, HTML

Christian Kästner  Vincent Hellendoorn
HW3 Reflections
Today

● GUI Design
  ○ Concepts, strategies
  ○ Practical application in HTML, CSS, JS

● Dynamic Web Pages
  ○ Client/Server communication
  ○ Backend architecture
How To Make This Happen?

- Have experience testing and analyzing your software
- Understand principles of concurrency and distributed systems

See a more detailed list of learning goals describing what we want students to know or be able to do by the end of the semester. We evaluate whether learning goals have been achieved through assignments and exams.

Coordinates

Tu/Th 11:50 - 1:10 p.m. In DH 2316

Christian Kaestner, kaestner@cs.cmu.edu, TCS 345, office hours Friday 11:30-1 pm (see calendar)

Vincent Heliendoorn, TCS 320, office hours Tuesdays 9am-11am (see calendar)

Our TAs also provide an additional 16th of office hours each week, usually in TCS 310, see details in the calendar.

The instructors have an open door policy: if the instructors’ office doors are open and no-one else is meeting with us, we are happy to answer any course-related questions. Also, feel free to email us for appointments.

Course Calendar

[Calendar grid with dates and times highlighted]

10:00 Ye Oh (Online) 11:00 Lecture
09:00 Recitation A 10:10 Recitation B 11:15 Recitation C
11:30 Christian Oh 16:00 Zhiheng Ch 19:00 Esther Ch
14:00 Online OH 15:00 Kevin Oh
09:00 Vincent Oh 11:00 Lecture
10:10 Recitation A 11:15 Recitation C
16:00 Zhiheng Ch 19:00 Esther Ch
15:00 Kevin Oh
09:00 Vincent Oh 11:00 Lecture
10:10 Recitation A 11:15 Recitation C
16:00 Zhiheng Ch 19:00 Esther Ch
15:00 Kevin Oh
09:00 Vincent Oh 11:00 Lecture
10:10 Recitation A 11:15 Recitation C
16:00 Zhiheng Ch 19:00 Esther Ch
15:00 Kevin Oh
09:00 Vincent Oh 11:00 Lecture
10:10 Recitation A 11:15 Recitation C
16:00 Zhiheng Ch 19:00 Esther Ch
15:00 Kevin Oh
Why not plaintext?

---

1. 17-214 Fall 2021
2. 17-214 Fall 2021
3. Principles of Software Construction
4. Object, Design, and Concurrency
5. Overview
6. Software engineers today are less likely to design data structures and algorithms from scratch and more likely to build systems from library and framework components. In this course, students engage with concepts related to the construction of software systems at scale, building on their understanding of the basic building blocks of data structures, algorithms, program structures, and computer
7. The course covers technical topics in four areas: (1) concepts of design for complex systems, (2) object-oriented programming, (3) static and dynamic analysis for programs, and (4) concurrent and distributed software. Student assignments involve engagement with complex software such as distributed massively multiplayer game systems and frameworks for graphical user
8. Interaction.
9. Update for Fall 2021: We are planning several changes to the course for the Fall 2021 semester. A key change is that we will teach the course with multiple programming languages. We will cover multiple languages in the lecture, but will expect students to focus on one language in assignments. When signing up, please choose a section for Java or JavaScript/TypeScript.
10. After completing this course, students will:
11. 17-214 Fall 2021: 1:19 p.m. in DH 2315
12. Be comfortable with object-oriented concepts and with programming in the Java or Javascript language
13. Have experience designing medium-scale systems with patterns
14. Understand principles of concurrency and distributed systems
15. See a more detailed list of learning goals describing what we want students to know or be able to do by the end of the semester. We evaluate whether learning goals have been achieved through assignments and exams.
16. Christian Koneker, kawastaw@cmu.edu, TCSS 308, office hours Friday 11:30-1pm (see calendar)
17. Vincent Hellendon, TCSS 308, office hours Tuesdays 9am-11am (see calendar)
18. Office hours also provide an additional 16 hours of office hours each week, usually in TCSS 308, see details in the calendar.
19. The instructors have an open door policy: if the instructors' office doors are open and no one else is meeting with us, we are happy to answer any course-related questions. Also, feel free to email us for appointments.
20. Course Calendar
21. Schedule
22. We are planning significant changes to the course this semester. The schedule below is a rough draft of our plans, but likely to change.
Why not a Doc?
GUI Design: what do we want?

- Nested Elements
- Style Vocabulary
- Interactivity
GUI Design: what do we want?

- Nested Elements
  - HTML
- Style Vocabulary
  - CSS
- Interactivity
  - JavaScript
Anatomy of an HTML Page

Predefined elements

Root*
Header
Body

Technically, ‘document’ is the root with HTML its only child
Anatomy of an HTML Page

Nested elements

● Sizing
● Attributes
● Text

**Overview**

Software engineers today are less likely to design data structures and algorithms from scratch and more likely to build systems from library and framework components. In this course, students engage with concepts related to the construction of software systems at scale, building on their understanding of the basic building blocks of data structures, algorithms, program structures, and computer structures. The course covers technical topics in four areas: (1) concepts of design for complex systems, (2) object oriented programming, (3) static and dynamic analysis for
Anatomy of an HTML Page

Many GUIs are trees

- Nested elements, recursively
- Some fixed positions (html, body)
Anatomy of an HTML Page

Many GUIs are trees

- Nested elements, recursively
- Some fixed positions (html, body)

How to implement this?

[Diagram showing the Document Object Model (DOM) with a tree structure, including elements such as `<html>`, `<head>`, `<body>`, `<title>`, `<h1>`, `<a>` with attributes and text content.]
The composite pattern

● Problem: Collection of objects has behavior similar to the individual objects

● Solution: Have collection of objects and individual objects implement the same interface

● Consequences:
  ○ Client code can treat collection as if it were an individual object
  ○ Easier to add new object types
  ○ Design might become too general, interface insufficiently useful
Another composite pattern example

```java
public interface Expression {
    double eval(); // Returns value
}

public class BinaryOperationExpression implements Expression {
    public BinaryOperationExpression(BinaryOperator operator,
                                      Expression operand1, Expression operand2);
}

public class NumberExpression implements Expression {
    public NumberExpression(double number);
}
```
Composite

- Elements can contain elements
  - With restrictions
  - Need to deal with style, interaction

- In JS: HTMLElement
  - With child-classes e.g. HTMLDivElement, HTMLBodyElement
  - Navigation:
    - getElement*: locate by tag name, id, class, etc.
    - next/prev(Element)Sibling
    - childNodes, parent
A few Tags

- `<html>`
  - The root of the visible page
- `<head>`
  - Stores metadata, imports
- `<p>`
  - A paragraph
- `<button>`
  - Attributes include `name`, `type`, `value`
- `<div>`
  - Generic section -- very useful
- `<table>`
  - The obvious
- Many more; dig into a real page!
Style

Not only leaf-nodes have an appearance
Style

Tags come with inherent & customizable style

● **Inherent:**
  ○ `<div>` is a `block` (full-width, with margin)
  ○ `<span>` is in-line
  ○ `<h1>` is large

● **Customizable: add and override styles**
  ○ Change font-styles, margins, widths
  ○ Modify groups of elements
Style: CSS

- Cascading Style Sheets
  - Reuse: styling rules for tags, classes, types
  - Reuse: not just at the leaves!

```html
<span style="font-weight:bold">Hello again!</span>
```

vs.

```html
<style type="text/css">
  span {
    font-family: arial
  }
</style>
```
Style: CSS

● Cascading Style Sheets
  ○ Reuse: styling rules for tags, classes, types
  ○ Reuse: not just at the leafs!

● What if there are conflicts?

   <div style="font-weight:normal">
     <span style="font-weight:bold">Hello again!</span>
   </div>

○ Lowest element wins*

*Technically, there’s a whole scoring system
Style: CSS

- Cascading Style Sheets
  - Reuse: styling rules for tags, classes, types
  - Reuse: not just at the leafs!
- What if there are no conflicts?

```html
<div style="font-family:arial">
  <span style="font-weight:bold">Hello again!</span>
</div>
```

- How would you implement this?
Style: CSS

What is happening here?
Decorator

What is happening here?

● To compute the style of an element:
  ○ Apply its tag-default style
  ○ Wrap in added style rules (tag-specific or general)
    ■ Text: font-family, weight, etc.
  ○ Inherit parents’ style
    ■ Conflicts lead to overrides

● Makes themes really powerful

Technically, HTML is streamed top-to-bottom; CSS works bottom-up
CSS: classes

Let’s not repeat custom style

● Use any nr. of class label(s)
● Class styles get added
● Facilitates reuse

How would you implement this?
Strategy or Observer?
Strategy or Observer?

Either could apply

- Both involve callback
- **Strategy:**
  - Typically single
  - Often involves a return
- **Observer:**
  - Arbitrarily many
  - Involves external updates
Interactivity

A GUI is more than a document

- How do we make it “work”?

Hi there!
Hello again!
Click me
Actions: JavaScript

- Key: event listeners (what’s that pattern?)
- (frontend) JS is highly event-driven
  - Respond to window `onLoad` event, content loads (e.g., ads)
  - Respond to clicks, moves
Observer Pattern

● Manages publishers and subscribers
  ○ Here, button publishes its ‘click’ events
  ○ ‘buttonClicked’ subscribes to 1+ updates

● Flexibility and Reuse
  ○ Multiple observers per element
  ○ Shared observers across elements
Step Back

- What is our website now?
  - Layout, style, interaction
  - What is missing?
Static Web Pages

- Delivered as-is, final
  - Consistent, often fast
  - Cheap, only storage needed
- “Static” a tad murky with JavaScript
  - We can still have buttons, interaction
  - But it won’t “go” anywhere -- the server is mum

https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview#anatomy_of_a_dynamic_request
Static Web Pages

● Delivered as-is, final
  ○ Consistent, often fast
  ○ Cheap, only storage needed

● Maintain with *static website generators*
  ○ Or you’ll be doing a lot of copying
  ○ Coupled with themes => rapid development, deployment
  ○ Quite popular, e.g. hosting on GH Pages
Static Web Pages

● But …
  ○ No persistence (at least, not obviously)
  ○ No customizability (e.g., accounts)
  ○ No communication (payment, chat, etc)
  ○ Realistically, no intensive jobs
Dynamic Web Pages

● **Client/Server**
  ○ Someone needs to answer the website’s calls
    ■ Doesn’t need to be us!
  ○ Host a [webserver](#)
    ■ Serves pages, handles calls
    ■ For static pages too!

● We’ll show you more tomorrow (Wednesday)
Web Servers

● Communicate via HyperText Transfer Protocol
  ○ URL (the address)
  ○ Method:
    ■ GET: retrieve data. Parameters in URL `...?key=value&key2=value2` and message body
    ■ POST: store/create data. Parameters in request body
    ■ Several more, rarely used
  ○ Responses:
    ■ Status Code. We all know 404. 2XX family is OK.
    ■ And possible data. E.g., entire HTML page.
Web Servers

- Communicate via HyperText Transfer Protocol
  - URL (the address)
  - Method:
    - GET: retrieve data. Parameters in URL `...?key=value&key2=value2` and message body
    - POST: store/create data. Parameters in request body
    - Several more, rarely used
  - Responses:
    - Status Code. We all know 404. 2XX family is OK.
    - And possible data. E.g., entire HTML page.
  - POST makes no sense for static sites!
  - As do GETs with parameters
Web Servers

Dynamic sites can do more work

https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview#anatomy_of_a_dynamic_request
AJAX

● Originally: “Asynchronous JavaScript and XML”
  ○ Updates parts of a page dynamically
  ○ Sends XMLHttpRequests with a callback
  ○ On return, check the code; handle success and failure.
  ○ Asynchronous, naturally decouples backend from UI
AJAX

● Originally: “Asynchronous JavaScript and XML”
  ○ Updates parts of a page dynamically
  ○ Sends XMLHttpRequests with a callback
  ○ On return, check the code; handle success and failure.
  ○ Asynchronous, naturally decouples backend from UI

● Slowly being phased out
  ○ Replace with `fetch`, which uses… Promises
    ■ More next week
How to Web App?

- Let’s avoid generating HTML from scratch on every call
  - Map requests to handler code
    - Fetch data, process
  - Generate and return HTML

- Historically: PHP
  - Modifies HTML pages server-side on request; strong ties to SQL

```php
<?php
    // The global $_POST variable allows you to access the data sent with the POST method by name
    // To access the data sent with the GET method, you can use $_GET
    $say = htmlspecialchars($_POST['say']);
    $to = htmlspecialchars($_POST['to']);

    echo $say, ' ', $to;
?
```
How to Web App?

● Let’s avoid generating HTML from scratch on every call
  ○ Map requests to handler code
    ■ Fetch data, process
  ○ Generate and return HTML

● Or use a framework
  ○ Python: Flask, Django
  ○ NodeJS: Express
  ○ Spring for Java
  ○ Many others, differences in weight, features
Model-View-Controller (MVC)

https://overiq.com/django-1-10/mvc-pattern-and-django/
MVC is ubiquitous

Separates:

- **Model**: data organization
  - Interface to the database
- **View**: data representation (typically HTML)
  - Often called *templates* in web-dev; “view” is a bit overloaded
- **Controller**: intermediary between client and model/view
  - Typically asks model for data, view for HTML
Core implementation vs. GUI

● Core implementation: application logic
  ○ Computing some result, updating data

● GUI
  ○ Graphical representation of data
  ○ Source of user interactions

● Design guideline: *avoid coupling the GUI with core application*
  ○ Multiple UIs with single core implementation
  ○ Test core without UI
Separating application core and GUI

- Reduce coupling: do not allow core to depend on UI
- Create and test the core without a GUI
  - Use the Observer pattern to communicate information from the core (Model) to the GUI (View)
Summary

● GUIs are full of design patterns
  ○ Helpful for reuse, delegation in complex environments

● Covered the basics of HTML, CSS, JS, servers
  ○ Needed for dynamic web pages
  ○ Decouple the GUI; architect your backend
  ○ A lot more to learn (security, performance, privacy), but this will do

● You will build this
  ○ At a small scale