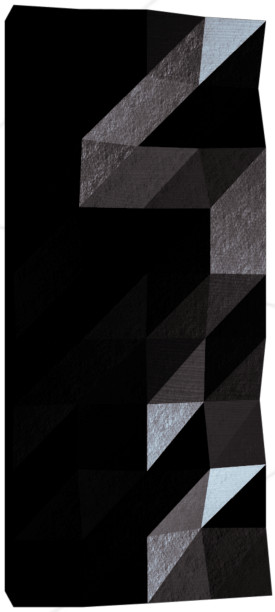

Structured Design

sd03

Summary: In this module you will apply what you learned on structured design

Version: 4.2

Contents

Chapter I

Preamble

Hello, and welcome *back* to your first Structured Development workshop!

This workshop is part of a mini-course for *42* students curated by *Prof. Enrico Vicario* and *Dr. Nicolò Pollini*. Throughout this course, you'll be introduced to a programming methodology that originated in the 1960s and has since evolved in tandem with the C language.

Structured Development is built on three key pillars:

- **Structured Programming**: This concerns the act of writing code in a clean, readable, and maintainable way. It emphasizes avoiding common pitfalls and code smells that lead to bugs and wasted time, especially during debugging.
- **Structured Design**: Good software design means organizing your codebase so that each function has a clear, well-defined purpose. The goal is to prevent situations where small changes ripple through the entire system, forcing you to rewrite large portions of code (yes, we're looking at you, *spaghetti code*).
- **Structured Analysis**: This is the most abstract level. It involves designing the architecture of an entire software system, especially when highly complex, based on its requirements and the flow of information needed to fulfill them.

`Remember: Code is not just written to work, it's written to be used. If it's worth using, it's worth writing for someone else to read, understand, and maintain. You never write code *just* for yourself.`

Chapter II

Introduction

Structured Design, as laid out by Edward Yourdon and Larry Constantine in their seminal 1979 book, is a systematic methodology for designing software systems with a strong emphasis on modularity and data flow. It aims to create highly maintainable, flexible, and efficient programs by breaking down complex systems into smaller and manageable components.

There are two fundamental metrics for evaluating the quality of a design:

- **Coupling**: Refers to the degree of interdependence between modules. Structured Design advocates for *low coupling*, meaning modules should be as independent as possible, communicating through well-defined, minimal interfaces. This reduces the impact of changes in one module on others.
- **Cohesion**: Refers to the degree to which elements within a module belong together. Structured Design promotes *high cohesion*, meaning a module should have a single, well-defined purpose and all its elements should contribute to that purpose. This makes modules easier to understand, maintain, and reuse.

In the **June 13** lecture, you have received some more in-depth guidance from Prof. Vicario on Structured Design. Software design is a discipline quite distinct from programming. While programming focuses on building something that works reliably and adheres strictly to given constraints, **software design often involves fewer rigid requirements and greater freedom in implementation decisions**. Structured Design emerged precisely to help navigate this increased freedom, enabling more educated choices that go beyond intuition alone.

Based on what you've learned in this lesson, we encourage you to revisit the exercises from the corresponding subject. This time, **try applying the concepts of Structured Design**.
You're not required to make any changes if you don't feel they're necessary, we simply ask that you **approach the exercises with an open mind and see how it goes**. At the end of the course, you-ll receive feedback on *both* versions. You're encouraged to explore freely: discuss ideas with your classmates, search for insights online, and feel free to use AI tools if they help you reason better.

However, a word of caution about **Large Language Models (LLMs)** like ChatGPT, Gemini, and similar tools:

Researchers (including us) have studied the impact of over-relying on AI in learning-driven development, and the findings suggest that, while LLMs can boost your short-term performance, **overusing them can hinder your long-term growth**. The knowledge and skills you build now are what your future success depends on.
Here's our recommended approach:

- **Start by solving the problem on your own.**
- **If you get stuck, ask your peers or look for help online.**
- **If you're still stuck, and no one can help, use LLMs to understand *how* to approach the problem**, but don't let them solve it for you, unless you're certain that solving this particular problem is not important to your learning journey.

Chapter III

General instructions

- This document contains **7 C programming exercises**, arranged in **increasing order of difficulty**. While you're free to tackle them in any order, we recommend progressing sequentially to get the most out of the experience.
- These exercises are **not meant to be completed in full by everyone**, so don't be discouraged if you struggle early on. This course brings together students with a wide range of experience, and the more advanced exercises are designed to challenge even seasoned programmers.
- **Tomorrow, we'll release the third set of exercises (Structured Analysis)**. So focus on doing your best with today's set, and try to apply what you learned during the lecture.

Chapter IV

Evaluation criteria

Unlike in previous experiences, you will be evaluated not only on whether your solution works and meets the requirements, but also on the **reasoning behind your choices**. In fact, this is the primary focus of the evaluation. *(No pressure, though)*
Here are a few important notes:

- **Your code must compile.** Minor oversights or non-critical errors may be tolerated, as long as the core functionality is preserved.
- **You're not required to follow the 42 Norm**, but it's a good starting point if you're unsure about what constitutes clean and readable code.

`Note: This is the first version of this document, so it may contain errors or inaccuracies. If you spot any issues, please let the staff know as soon as possible. Thank you!`

Chapter V

Exercise 00 : movie_planner

	Exercise 00
Exercise 00 : movie_planner	
Turn-in directory : $ex00/$	
Files to turn in : `movie_planner.c, movie_planner.h`	
Allowed functions : `free`	

You're planning a chill movie night with your friends. The goal is to orchestrate the process of collecting your friends' preferences, searching for matching movies, and building a movie plan for the night.

Your task is to design the function hierarchy to build the plan using the following prototypes:

```
// Returns user preferences. Returns NULL on failure.
struct Preferences *get_user_preferences(void);

// Returns a list of movies matching the given preferences. Returns NULL on failure.
struct MovieList *find_movies(struct Preferences *prefs);

// Returns a movie night plan from the given list. Returns NULL on failure.
struct Plan *build_plan(struct MovieList *list);
```

You don't need to implement the above functions, focus on designing a clean control structure without worrying about actual implementations.
The function must be formatted as follows:

```
struct Plan *create_movie_night_plan(void);
```

Chapter VI

Exercise 01 : food_order

	Exercise 01
Exercise 01 : food_order	
Turn-in directory : $ex01/$	
Files to turn in : `food_order.c, food_order.h`	
Allowed functions : `free`	

You're building the backend for a food delivery app. A critical task is to process an incoming order. The logic changes depending on whether the restaurant is currently open or closed. If it's open, you create a standard order confirmation. If it's closed, you create a pre-order confirmation for the next day.

Your goal is to implement the control logic that, given an order request, checks the restaurant's status, prepares the appropriate confirmation, and sends a notification.

Your task is to design the function hierarchy to build the app using the following prototypes:

```
// Checks if the restaurant is open based on the order. Returns 1 if open, 0 if closed.
int check_restaurant_status(struct OrderRequest *request);

// Creates a confirmation for an immediate order. Returns NULL on failure.
struct OrderConfirmation *create_standard_confirmation(void);

// Creates a confirmation for a future (pre-order). Returns NULL on failure.
struct OrderConfirmation *create_preorder_confirmation(void);

// Sends the confirmation to the user.
void send_confirmation_notification(struct OrderConfirmation *confirmation);
```

You don't need to implement the above functions, focus on designing a clean control structure without worrying about actual implementations.
The function must be formatted as follows:

```
int process_food_order(struct OrderRequest *request);
```

Chapter VII

Exercise 02 : music_organizer

	Exercise 02
Exercise 02 : music_organizer	
Turn-in directory : $ex02/$	
Files to turn in : `music_organizer.c, music_organizer.h`	
Allowed functions : `free`	

You're a music enthusiast and want to organize your vast digital music library. Your goal is to automate the process of scanning music files, categorizing them, and updating a central music database.

Your task is to design the function hierarchy to organize the music library using the following prototypes:

```
// Creates and returns a new music library. Returns NULL on fail
struct MusicLibrary *create_music_library();

// Scans a directory for music files. Returns a NULL terminated array of filenames.
const char **scan_directory(const char *directory_path);

// Processes a single music file. Returns a data structure representing the processed file.
struct MusicFile *process_music_file(const char *directory_path, const char *filename);

// Updates the music library with the processed music file information.
void update_music_library(struct MusicLibrary *library, struct MusicFile *song);
```

You don't need to implement the above functions, focus on designing a clean control structure without worrying about actual implementations.
The function must be formatted as follows:

```
struct MusicLibrary *organize_music_library(const char *directory_path);
```

Chapter VIII

Exercise 03 : workout_scheduler

	Exercise 03
Exercise 03 : workout_scheduler	
Turn-in directory : $ex03/$	
Files to turn in : `workout_scheduler.c, workout_scheduler.h`	
Allowed functions : `None`	

You're building a workout scheduler for a fitness tracking app. The system fetches a user's data, builds a preliminary workout plan, refines it with the user's preferences, and then finalizes a complete schedule for a certain number of days. Each day includes personalized exercises and motivational tips.

Your task is to design the function hierarchy to process the workout scheduler using the following prototypes:

```
// Returns a new UserData (mocked). Returns NULL on failure.
struct UserData *get_user_data(char *username);

// Build a base WorkoutPlan from raw user data. Returns NULL on failure.
struct WorkoutPlan *build_base_plan(struct UserData *data);

// Optionally refine an existing plan. Returns the same pointer (or a new one) or NULL on failure.
struct WorkoutPlan *refine_plan(struct WorkoutPlan *plan, struct UserData *data);

// Determine how many days the workout schedule should span. Returns positive int, or <=0 on failure.
int determine_duration(struct WorkoutPlan *plan);

// Assign daily exercises for the next day into the plan.
void assign_daily_exercises(struct WorkoutPlan *plan, int day);

// Assign daily tips for the next day into the plan.
void assign_daily_tips(struct WorkoutPlan *plan, int day);

// Free functions for cleanup.
void free_user_data(struct UserData *data);
void free_workout_plan(struct WorkoutPlan *plan);
```

You don't need to implement the above functions, focus on designing a clean control structure without worrying about actual implementations.

The function must be formatted as follows:

```
struct WorkoutPlan *create_workout_schedule(char *username);
```

Chapter IX

Exercise 04 : recipe_generator

	Exercise 04
Exercise 04 : recipe_generator	
Turn-in directory : $ex04/$	
Files to turn in : `recipe_generator.c, recipe_generator.h`	
Allowed functions : `None`	

You are building an interactive recipe generator that creates recipes based on the user's current ingredients and taste profile, repeating the generation process until the user approves the result. The program should:

1. Fetch the current fridge ingredients.
2. Fetch the user's taste profile.
3. Iteratively generate a recipe candidate:
   - Produce a new recipe.
   - Ask for user approval.
   - Repeat until approved.
4. Return the final approved recipe.

Your task is to design this control flow, handling errors and resource cleanup, by using the following prototypes:

```
// Fetch current ingredients from fridge. Returns a new Ingredients* or NULL on failure.
struct Ingredients *get_current_ingredients(void);

// Fetch user taste profile. Returns a new TasteProfile* or NULL on failure.
struct TasteProfile *get_user_taste_profile(void);
```

```
// Generate a new recipe candidate based on ingredients & taste. Returns a new Recipe* or NULL on
    failure.
struct Recipe *create_recipe(struct Ingredients *ingredients, struct TasteProfile *taste);

// Ask user approval for the given recipe. Returns 1 if approved, 0 otherwise.
int get_user_approval(struct Recipe *recipe);

// Free functions for cleanup:
void free_ingredients(struct Ingredients *ing);
void free_taste_profile(struct TasteProfile *tp);
void free_recipe(struct Recipe *recipe);
```

You don't need to implement the above functions, only focus on structuring the control flow.

The function must be formatted as follows:

```
struct Recipe *create_custom_recipe(void);
```

Chapter X

Exercise 05 : playlist_creator

	Exercise 05
Exercise 05 : playlist_creator	
Turn-in directory : $ex05/$	
Files to turn in : `playlist_creator.c, playlist_creator.h`	
Allowed functions : `None`	

You are building a personalized playlist generator for a music app. The program should:

1. Analyze the user's mood.
2. Build initial filter settings based on default filters.
3. Refine filters according to mood variations.
4. Assemble a final playlist using mood and filter settings:
   - Depending on filter characteristics, fetch a popular or niche song.
   - Combine the chosen song into a mood playlist

Your task is to design the function hierarchy to build the playlist using the following prototypes:

```
// Analyze user mood. Returns a new MoodSettings or NULL on failure.
struct MoodSettings *analyze_user_mood(void);

// Return default filter settings. Returns a new FilterSettings or NULL on failure.
struct FilterSettings *default_filters(void);

// Return the number of mood variations for refinement.
int get_mood_variations(struct MoodSettings *mood);

// Refine the current filter settings. Returns the same or a new FilterSettings, or NULL on failure.
```

```
struct FilterSettings *refine_filters(struct FilterSettings *current);

// Check if filters require a popular or niche song. Returns nonzero if popular.
int filters_require_popular_song(struct FilterSettings *filters);

// Fetch a popular song. Returns a new SongData or NULL on failure.
struct SongData *fetch_popular_song(void);

// Fetch a niche song. Returns a new SongData or NULL on failure.
struct SongData *fetch_niche_song(void);

// Combine a song into a playlist given mood settings. Returns a new Playlist or NULL on failure.
struct Playlist *combine_with_mood_playlist(struct SongData *song, struct MoodSettings *mood);

// Free functions for cleanup
void free_mood_settings(struct MoodSettings *mood);
void free_filter_settings(struct FilterSettings *filters);
void free_song_data(struct SongData *song);
void free_playlist(struct Playlist *playlist);
```

You don't need to implement the above functions, only focus on structuring the control flow.

The function must be formatted as follows:

```
struct Playlist *create_playlist(void);
```

Chapter XI

Exercise 06 : task_scheduler

	Exercise 06
Exercise 06 : task_scheduler	
Turn-in directory : $ex06/$	
Files to turn in : `task_scheduler.c, task_scheduler.h`	
Allowed functions : `None`	

You are building a simulated task scheduler for a multicore CPU system. The program should:

1. Profile the provided task list to evaluate load and I/O patterns.
2. Compute scheduling priorities for each task.
3. Assign tasks to CPU cores using the computed priorities.

Your task is to design a well-layered control flow that delegates these responsibilities using the following prototypes:

```
// Analyze task load and I/O behavior. Returns a new TaskProfile or NULL on failure.
struct TaskProfile *profile_tasks(struct TaskList *tasks);

// Compute a priority map from task profiling. Returns a new PriorityMap or NULL on failure.
struct PriorityMap *compute_priorities_mock(struct TaskProfile *profile);

// Return the number of available CPU cores.
int get_available_core_count(void);

// Select the best task to assign next based on the profile and the priority map. Returns a task ID or
      -1 on failure.
int select_best_task_for_current_priorities(struct TaskProfile *profile, struct PriorityMap *
    priorities);

// Create a ScheduleResult structure for the specified number of cores. Returns NULL on failure.
struct ScheduleResult *create_schedule_result(int core_count);
```

```
// Update the result schedule with a task assignment.
void update_schedule_entry(struct ScheduleResult *result, int core_id, int task_id);

// Free functions for cleanup
void free_task_profile(struct TaskProfile *profile);
void free_priority_map(struct PriorityMap *priorities);
void free_schedule_result(struct ScheduleResult *result);
```

You don't need to implement the above functions, only focus on structuring the control flow.

The function must be formatted as follows:

```
struct ScheduleResult *schedule_tasks(struct TaskList *tasks);
```

Chapter XII

Submission and peer evaluation

Create a personal repository named:
42xunifi-structured-development-2025-<your-intra-login>
Share it with the staff and organize the exercises as follows:

```
sd00/
 ex00/
 ex01/
 ex02/
 ...
sd01/
 ex00/
 ex01/
 ...
sd02/
 ex00/
 ...
```

You can share your repository by filling out this form

- **Update your repository regularly**, and make sure to **push all completed exercises before the deadline**.
- **This workshop's deadline is June 20**. Any changes made after this date **will not be considered** for assessment purposes.
- This exercise follows the **42 methodology**, so **peer-to-peer collaboration is allowed and strongly encouraged**.
- **Formal peer evaluations via Intra will not be available**, but you are still welcome to ask your peers for feedback on your work.

Our feedback won't be immediate, so to help you track your progress, we recommend using **CCCC (C and C++ Code Counter)**, a command-line, open-source tool designed for analyzing source code in C, C++, and (in more recent versions) Java. Its primary purpose is to generate reports on various software metrics from the code it processes, such as:

- *McCabe's Cyclomatic Complexity*: This metric measures the complexity of a program's control flow. A higher cyclomatic complexity often indicates code that is harder to test, understand, and maintain.
- *Measures of module fan-out/fan-in*: These design metrics describe the dependencies between modules and are closely linked to the concept of coupling.

- *Chidamber & Kemerer and Henry & Kafura*: These metrics are frequently associated with object-oriented design principles, such as cohesion.

Combined with SonarQube, these metrics from CCCC will help you better understand your progress with Structured Design. Since CCCC is a command-line tool, it should be *your cup of tea*, thus we won't be providing installation instructions here.

Chapter XIII

Setting Up SonarQube (SonarCloud) with GitHub

1. **Create a public GitHub repository** for the course, named: `42xunifi-structured-development-2025-<your-intra-login>`.
2. Go to SonarCloud and click **"Start now."**
3. Sign up using your **GitHub account**.
4. When prompted, **import an organization**. You can use `"42"` or your GitHub username.
5. **Authorize access only to selected repositories**, and choose the one you created in step 1.
6. Choose a **Name** and a **Key** for your organization (your username is usually fine for both).
7. Select the **Free Plan** when prompted.
8. Click **"Create Organization."**
9. Select the previously created repository and click **"Set Up."**
10. For code technology, choose **"Previous version."**
11. Click **"Create Project."**
12. *Profit.*