Software Engineering

Ionuț Cicio

12/01/2026

The latest version of the `.pdf` and the referenced material can be found at the following link: https://github.com/CuriousCI/software-engineering

Contents

1 Model based software design

Software projects require **design choices** that often can't be driven by experience or reasoning alone. That's why a **model** of the project is needed to compare different solutions before committing to a design choice.

1.1 The *"Amazon Prime Video"* dilemma

If you were tasked with designing the software architecture for Amazon Prime Video, which choices would you make? What if you had the to keep the costs minimal? Would you use a distributed architecture or a monolith application?

More often than not, monolith applications are considered more costly and less scalable than the counterpart, due to an inefficient usage of resources. But, in a recent article, a Senior SDE at Prime Video describes how they *"**reduced the cost** of the audio/video monitoring infrastructure by **90%**"* [1] by using a monolith architecture.

There isn't a definitive way to answer these type of questions, but one way to go about it is building a model of the system to compare the solutions. In the case of Prime Video, *"the audio/video monitoring service consists of three major components:"* [1]

- the *media converter* converts input audio/video streams
- the *defect detectors* analyze frames and audio buffers in real-time
- the *orchestrator* controls the flow in the service

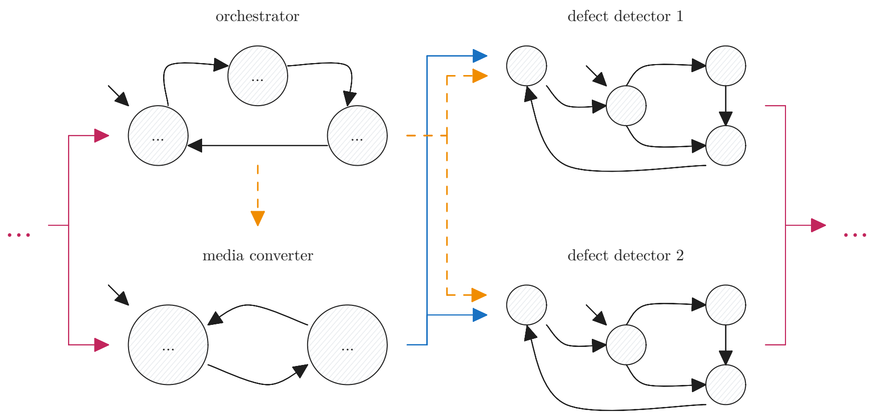

Figure 1: audio/video monitoring system process

To answer questions about the system, it can be simulated by modeling its components as **Markov decision processes**.

1.2 Probabilistic software modeling

1.2.1 Markov chains

Definition 1 (Markov chain)**.** A Markov chain M is a pair (S, p) where

- S is the set of states
- $p : S \times S \to [0, 1]$ is the transition probability

The function p is such that $p(s'|s)$ is the probability to transition from state s to state s'. For it to be a probability it must follow Equation 1

$$\forall s \in S \quad \sum_{s' \in S} p(s'|s) = 1 \tag{1}$$

A Markov chain (or Markov process) is characterized by *memorylesness* (also called the Markov property), meaning that predictions on future states can be made solely on the present state of the Markov chain and predictions are not influenced by the history of transitions that led up to the present state.

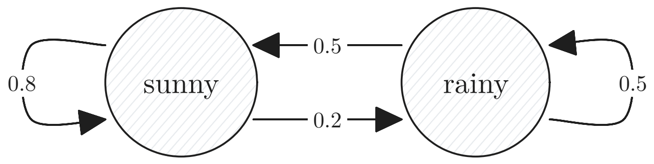

Figure 2: example Markov chain with $S = \{\texttt{rainy}, \texttt{sunny}\}$

If a given Markov chain M transitions at **discrete timesteps** (i.e. the time steps $t_1, t_2, \ldots$ are a countable) and the **state space** is countable, then it's called a DTMC (discrete-time Markov chain). There are other classifications for continuous state space and continuous-time.

p	sunny	rainy
sunny	0.8	0.2
rainy	0.5	0.5

Table 1: transition matrix of Figure 2

A Markov chain M can be written as a **transition matrix**, like the one in Table 1. Later in the guide it will be shown that implementing transition matrices, thus Markov chains, is really simple with the `<random>` library in `C++`.

1.2.2 Markov decision processes

A Markov decision process (MDP), despite sharing the name, is **different** from a Markov chain, because it interacts with an **external environment**.

Definition 2 (Markov decision process)**.** A Markov decision process M is conventionally a tuple (U, Y, X, p, g) where

- U is the set of input values
- Y is the set of output values
- X is the set of states
- $p : X \times X \times U \to [0, 1]$ is the transition probability
- $g : X \to Y$ is the output function

The same constrain in Equation 1 holds for MDPs, with an important difference: **for each input value**, the sum of the transition probabilities for that input value must be 1.

$$\forall x \in X, u \in U \quad \sum_{x' \in X} p(x'|x, u) = 1 \tag{2}$$

Where $p(x'|x, u)$ is the probability to transition from state x to state x' when the input is u.

Example 1 (Software development process)**.** The software development process of a company can be modeled as a MDP $M = (U, Y, X, p, g)$ where

- $U = \{\varepsilon\}$[1]
- $Y =$ euro $\times$ duration
- $X = \{x_0, x_1, x_2, x_3, x_4\}$

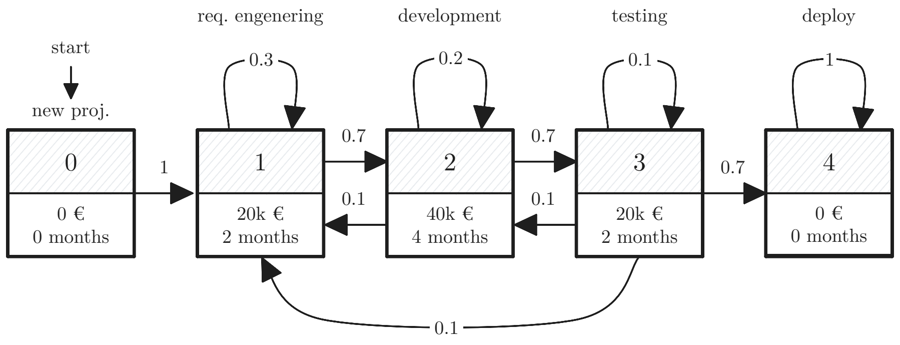

Figure 3: the model of a team's development process

$$g(x) = \begin{cases} (0, 0) & x \in \{x_0, x_4\} \\ (20000, 2) & x \in \{x_1, x_3\} \\ (40000, 4) & x \in \{x_2\} \end{cases}$$

ε	x_0	x_1	x_2	x_3	x_4
x_0	0	1	0	0	0
x_1	0	.3	.7	0	0
x_2	0	.1	.2	.7	0
x_3	0	.1	.1	.1	.7
x_4	0	0	0	0	1

[1]If U is empty M can't transition, at least 1 input is required, i.e. ε

Only one transition matrix is needed, as $|U| = 1$ (there is only one input value). If U had multiple input values, like $\{\mathsf{start}, \mathsf{stop}, \mathsf{wait}\}$, then multiple transition matrices would have been required, one for each input value.

1.2.3 Networks of Markov decision processes

Multiple MDPs can be connected into a network, and the network is itself a MDP that maintains the MDP properties (the intuition is there, but it's too much syntax for me to be bothered to write it).

Definition 3 (Network of MDPs)**.** Given a pair of Markov decision processes M_1, M_2 where

- $M_1 = (U_1, Y_1, X_1, p_1, g_1)$
- $M_2 = (U_2, Y_2, X_2, p_2, g_2)$

Let $M = (U, Y, X, p, g)$ be the network of M_1, M_2 such that

- TODO

1.3 Numerical analysis tips and tricks

1.3.1 Incremental mean

Given a set of values $X = \{x_1, ..., x_n\} \subset \mathbb{R}$ the mean value is defined as

$$\overline{x}_n = \frac{\sum_{i=1}^{n} x_i}{n} \tag{3}$$

$\overline{x}_n$ can be computed with the procedure in Listing 1.

```cpp
float traditional_mean(std::vector<float> values) {
    float values_sum =
        std::accumulate(values.begin(), values.end(), 0);

    return values_sum / (float)values.size();
}
```

Listing 1: `listings/mean.cpp`

The problem with this procedure is that, by adding up all the values before the division, the numerator could **overflow**, even if the value of $\overline{x}_n$ fits within the IEEE-754 limits. Nonetheless, $\overline{x}_n$ can be calculated incrementally.

$$\overline{x}_{n+1} = \frac{\sum_{i=1}^{n+1} x_i}{n+1} = \frac{\left(\sum_{i=1}^{n} x_i\right) + x_{n+1}}{n+1} = \frac{\sum_{i=1}^{n} x_i}{n+1} + \frac{x_{n+1}}{n+1} =$$
$$\frac{\left(\sum_{i=1}^{n} x_i\right)n}{(n+1)n} + \frac{x_{n+1}}{n+1} = \frac{\sum_{i=1}^{n} x_i}{n} \cdot \frac{n}{n+1} + \frac{x_{n+1}}{n+1} = \tag{4}$$
$$\overline{x}_n \cdot \frac{n}{n+1} + \frac{x_{n+1}}{n+1}$$

With this formula the numbers added up are smaller: $\overline{x}_n$, the mean, is multiplied by $\frac{n}{n+1} \sim 1$, and added up to $\frac{x_{n+1}}{n+1} < x_{n+1}$.

```cpp
float incremental_mean(std::vector<float> values) {
    float mean = 0;
    for (size_t n = 0; n < values.size(); n++) {
        mean =
            mean * ((float)n / (n + 1)) + values[n] / (n + 1);
    }

    return mean;
}
```

Listing 2: `listings/mean.cpp`

The examples in `listings/mean.cpp` show how the incremental computation of the mean gives a valid result, whereas the traditional procedure returns `Inf`.

1.3.2 Welford's online algorithm

In a similar fashion it could be faster and require less memory to calculate the standard deviation incrementally. Welford's online algorithm can be used for this purpose [2].

$$
\begin{gathered}
M_{2,n} = \sum_{i=1}^{n} (x_i - \overline{x}_n)^2 \\
M_{2,n} = M_{2,n-1} + (x_n - \overline{x}_{n-1})(x_n - \overline{x}_n) \qquad (5) \\
\sigma_n^2 = \frac{M_{2,n}}{n}
\end{gathered}
$$

Given M_2, if $n > 0$, the standard deviation is $\sqrt{\frac{M_{2,n}}{n}}$. The average can be calculated incrementally like in Section 1.3.1.

```cpp
void DataDistribution::insertDataPoint(real_t data_point) {
    real_t mean_prev_size =
        mean_ * (
            (real_t)number_of_data_points /
            (number_of_data_points + 1)
        ) +
        data_point / (number_of_data_points + 1);

    m_2__ +=
        (data_point - mean_) * (data_point - mean_prev_size);
    mean_ = mean_prev_size;
    number_of_data_points++;
}

real_t DataDistribution::mean() const { return mean_; }

real_t DataDistribution::stddev() const {
    return number_of_data_points > 0 ?
        sqrt(m_2__ / number_of_data_points) : 0;
}
```

Listing 3: `mocc/math.cpp`

1.3.3 Euler method

Many systems can be modeled via differential equations. When an ordinary differential equation can't be solved analitically the solution must be approximated. There are many techniques: one of the simplest ones (yet less accurate and efficient) is the forward Euler method, described by the following equation:

$$y_{n+1} = y_n + \Delta \cdot f(x_n, y_n) \qquad (6)$$

Where the function y is the solution to a problem like $\begin{cases} y(x_0)=y_0 \\ y'(x)=f(x,y(x)) \end{cases}$

Where $y(x_0) = y_0$ be the initial condition of the system, and $y' = f(x, y(x))$ be the known derivative of y. To approximate y a step Δ

is chosen (a smaller Δ results in a better approximation). Equation 6 can be intuitively explained like this: the value of y at each step is the previous value of y plus the value of its derivative y' multiplied by Δ. In Equation 6, y' is multiplied by Δ because when simulating one step of the system all the derivatives from x_n to x_{n+1} must be added up:

$$(x_{n+1} - x_n) \cdot f(x_n, y_n) = \Delta \cdot f(x_n, y_n) \tag{7}$$

Let's consider the example in Equation 8.

$$\begin{cases} y(x_0) = 0 \\ y'(x) = 2x \end{cases}, \quad \text{with } \Delta \in \left\{1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}\right\} \tag{8}$$

The following program approximates Equation 8 with different Δ values.

```cpp
/* y(x) = x^2 ⇒ y'(x) = 2x */
float y_prime(float x) { return 2 * x; }
```

Listing 4: `listings/euler.cpp`

```cpp
    for (size_t index = 0; index < DELTA_COUNT; index++) {
        float y = 0.0;
        const float delta = 1.0 / (float)(index + 1);
        for (float x = 0; x ≤ 10; x += delta) {
            results_files[index] << x << ' ' << y << std::endl;
            y += delta * y_prime(x);
        }
    }
```

Listing 5: `listings/euler.cpp`

The approximation in Figure 4 is close to x^2, but not very precise, however, error analysis is beyond this guide's scope.

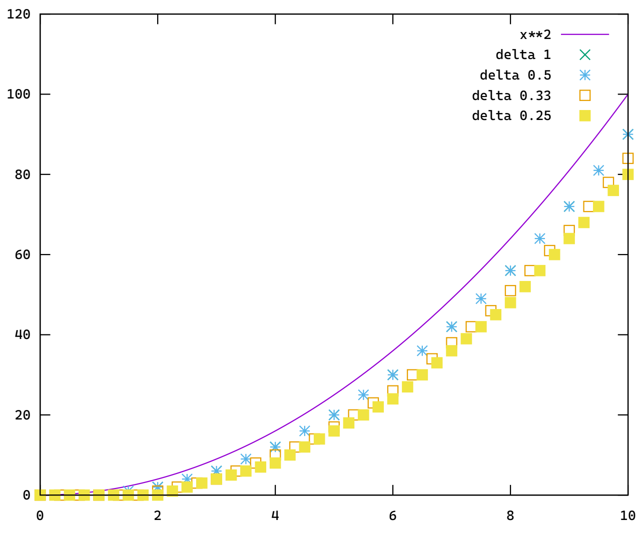

Figure 4: `public/euler.svg`

1.3.4 Monte Carlo method

Monte Carlo methods, or Monte Carlo experiments, are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. [3]

The underlying concept is to use randomness to solve problems that might be deterministic in principle [...] Monte Carlo methods are mainly used in three distinct problem classes: optimization, numerical integration, and generating draws from a probability distribution. [3]

Problem 1 (No budget left)**.** The following problem involves MyCAD™ the next generation **C**omputer **A**ided **D**rawing software. After a year of development, the remaining budget for MyCAD™ is only 550€; during the past year it has been observed that the cost to develop a new feature for MyCAD™ is described by the uniform distribution $\mathcal{U}(300€, 1000€)$. In order to choose whether to spend the reamining budget, find the probability that the next feature of MyCAD™ costs less than 550€.

```cpp
#include <iostream>

#include "listings.hpp" /* EXPERIMENTS = 1000000 */

int main() {
    /* A random number generator (or random engine) is
     * required. This function is explained in detail in
     * Section 2.2.1. */
    std::default_random_engine pseudo_random_engine =
        pseudo_random_engine_from_device();

    std::uniform_int_distribution<> rand_feature_cost(
        300, 1000
    );

    size_t number_of_experiments_less_than_550 = 0;
    for (size_t _ = 0; _ < EXPERIMENTS; _++) {
        /* Sample a random feature cost. */
        if (rand_feature_cost(pseudo_random_engine) < 550) {
            /* Yield 1 if the feature's cost is less than 550,
             * then add it to the numerator. */
            number_of_experiments_less_than_550++;
        }
    }

    /* Equation 11. */
    std::cout << (float)number_of_experiments_less_than_550 /
                    (float)EXPERIMENTS
              << std::endl;

    return EXIT_SUCCESS;
}
```

Listing 6: `listings/montecarlo.cpp`

The idea behind the Monte Carlo method is to execute a large number of **independent** experiments with the **same probability distribution** (i.i.d. experiments). Each experiment yields a value and, given the law of large numbers, the mean of the values yielded by the experiments tends to match to the mean value of the distribution as the number of experiments increases.

In the *"No budget left"* Problem the experiments can be modeled with a Bernoulli distribution, since either the next feature costs less than 550€ or not. The parameter p of the Bernoulli distribution is the probability which needs to be estimated.

Each experiment draws a uniform random number c between 300 and 1000 (the cost of the feature), and yields either 0 or 1 as described in Equation 9.

$$\begin{cases} 0 & c \geq 550€ \\ 1 & c < 550€ \end{cases} \tag{9}$$

This means that the parameter p of the Bernoulli distribution, which is the probability that the feature costs less than 550€, is calculated as

$$\begin{gathered} \frac{\text{number of experiments with value } 0 + \text{number of experiments with value } 1}{\text{total number of experiments}} \\ \overset{1.}{=} \\ \frac{\text{number of experiments with value } 1}{\text{total number of experiments}} \\ \overset{2.}{=} \\ \frac{\text{number of experiments with value less than } 550€}{\text{total number of experiments}} \end{gathered} \tag{10}$$

1. 0 is the identity element of the sum
2. by the definition in Equation 9

This type of calculation can be very easily distributed on a HPC cluster, and is generally an embarrassingly parallel problem [4], since each experiment is independent from the others.

2 C++ for modeling

This section covers the basics useful for the exam, assuming the reader already knows `C` and has some knowledge about `OOP`.

2.1 C++ prelude

`C++` is a strange language, and some of its quirks need to be pointed out to have a better understanding of what the code does in later sections.

2.1.1 Operator overloading

```cpp
#include <iostream>

struct Potato {
    float weight_g; // grams
};

struct Person {
    std::string name;
    float weight_kg; // kilograms

    void operator+=(const Potato &potato) {
        this->weight_kg += potato.weight_g / 1000;
    }

    void operator()() { std::cout << this->name << std::endl; }
};

int main() {
    Person pippo = {.name = "Pippo", .weight_kg = 102};
    Potato heavy_potato = {.weight_g = 352};

    std::cout << pippo.weight_kg << std::endl; /* 102 Kg */
    pippo += heavy_potato;
    std::cout << pippo.weight_kg << std::endl; /* 102.352 Kg */

    for (size_t _ = 0; _ < 5; _++) {
        pippo += heavy_potato;
    }

    std::cout << pippo.weight_kg << std::endl; /* 104.112 Kg */
    pippo(); /* Using the "()" operator on the object. */

    return EXIT_SUCCESS;
}
```

Listing 7: `listings/operator_overloading.cpp`

C++, like many other languages, allows the programmer to define how a certain operator should behave on an object (eg. `+`, `-`, `++`, `+=`, `<<`, `()`, `[]`, `etc ...`).

This feature is called **operator overloading**, and many languages other than C++ support it:

- in `Python` operator overloading is done by implementing methods with special names, like `__add__()` [5]
- in `Rust` it's done by implementing `Traits` associated with the operations, like `std::ops::Add` [6].
- `Java` and `C` do **not** support operator overloading

For example, `std::cout` is an instance of the `std::basic_ostream class`, which overloads the method "`operator<<()`" [7]; `std::cout << "Hello"` is a valid piece of code which prints on the standard output the string `"Hello"`, it does **not** do a bitwise left shift like it would in `C`.

2.2 Randomness in the standard library

The `C++` standard library offers tools to easily implement the Markov processes discussed in Section 1.2.1 and Section 1.2.2.

2.2.1 Randomness (random engines)

In `C++` there are many ways to generate random numbers [8]. Generally it's not recommended to use the `random()` function. It's better to use a random generator (like `std::default_random_engine`), because it's fast, deterministic (given a seed, the sequence of generated numbers is the same) and can be used with distributions. A `random_device` is a non deterministic generator: it uses a **hardware entropy source** (if available) to generate the random numbers.

```cpp
#include <iostream>
#include <random>

int main() {
    std::cout << random() << std::endl;

    std::random_device device_randomness_source; /* 1 */
    std::cout << device_randomness_source() << std::endl;

    auto seed = device_randomness_source(); /* 2 */

    std::default_random_engine /* 3 */
        pseudo_random_engine(seed);
    std::cout << pseudo_random_engine() /* 4 */ << std::endl;

    return EXIT_SUCCESS;
}
```

Listing 8: `listings/random.cpp`

The typical course of action is to instantiate a `random_device` `/* 1 */`, and use it to generate a seed `/* 2 */` for a `random_engine` `/* 3 */`.

Given that random engines can be used with distributions, they're really useful to implement MDPs. Random number generation is an example of overloading of `operator() /* 4 */`, like in Section 2.1.1.

From this point on, `std::default_random_engine` will be reffered to as `urng_t` (uniform random number generator type) for brevity.

```
#include <random>

/* The keyword "using" allows to create type aliases. */
using urng_t = std::default_random_engine;

/* The constructor of urng_t is called with parameter 190201. */
int main() { urng_t urng(190201); }
```

2.2.2 Probability distributions

Just the capability to generate random numbers isn't enough, these numbers need to be manipulated to fit certain needs. Luckly, `C++` covers most of them. To give an idea, the MDP in Figure 3 can be easily simulated with the code in Listing 10 below.

```
#include <iostream>

#include "listings.hpp"

int main() {
    /* As described in Section 2.2.1. */
    std::default_random_engine pseudo_random_engine =
        pseudo_random_engine_from_device();

    /* Transition matrix for MDP in Figure 3. */
    std::discrete_distribution<>
        markov_chain_transition_matrix[] = {
            {0, 1},
            {0, 0.3, 0.7},
            {0, 0.2, 0.2, 0.6},
            {0, 0.1, 0.2, 0.1, 0.6},
            {1},
        };

    const size_t HORIZON = 15, INITIAL_STATE = 0;

    size_t current_state = INITIAL_STATE;
    for (size_t _ = 0; _ < HORIZON; _++) {
        std::cout << current_state << std::endl;
        size_t next_state =
            markov_chain_transition_matrix[current_state](
                pseudo_random_engine
            );
        current_state = next_state;
    }
    return EXIT_SUCCESS;
}
```

Listing 10: `listings/markov_chain_transition_matrix.cpp`

2.2.2.1 Uniform discrete distribution (docs)

Problem 2 (Sleepy system)**.** To test the *sleepy system* S it's necessary to build a generator that sends a value v_i to S after waiting δ_i seconds after sending value v_{i-1} *(otherwise S is idle)*. The value of δ_i is an integer chosen with the uniform distribution $\mathcal{U}(20, 30)$.

The `C` code to compute δ_i would be `delta = 20 + rand() % 11;`, which is very **error prone**, hard to remember and has no semantic value.

In `C++` the same can be done in a simpler and cleaner way:

```
urng_t urng = pseudo_random_engine_from_device();
std::uniform_int_distribution<> random_delta(20, 30); /* 1 */
size_t delta = random_delta(urng); /* 2 */
```

The wait time δ_i can be easily generated without needing to remember any formula or trick `/* 2 */`. The distribution is defined once at the beginning `/* 1 */`, and it can be easily changed without introducing bugs or inconsistencies. It's also worth to take a look at a possible implementation of Problem 2 (with the addition that the i-th value sent is δ_i, meaning $v_i = \delta_i$), as it comes up very often in software models.

```
#include <iostream>

#include "listings.hpp"

int main() {
    std::default_random_engine pseudo_random_engine =
        pseudo_random_engine_from_device();

    std::uniform_int_distribution<> random_delta(20, 30);
    size_t delta = random_delta(pseudo_random_engine),
           next_send_time = delta;

    for (size_t time = 0; time < SIMULATION_HORIZON; time++) {
        if (time < next_send_time) {
            continue;
        }

        std::cout << delta << std::endl; /* v_i */
        delta = random_delta(pseudo_random_engine);
        next_send_time = time + delta;
    }
    return EXIT_SUCCESS;
}
```

Listing 11: `listings/time_intervals_generator.cpp`

The `uniform_int_distribution` has many other uses, for example, it could uniformly generate a random state in a MDP. Let `STATES_SIZE` be the number of states

```
uniform_int_distribution<> random_state(0, STATES_SIZE-1 t1);
```

`random_state` generates a random state when used. Be careful! Remember to use `STATES_SIZE-1` ①, because `uniform_int_distribution` is inclusive. Forgettig `-1` can lead to very sneaky bugs, like random segfaults at different instructions. It's very hard to debug unless using `gdb`. The `uniform_int_distribution` can also generate negative integers, for example $z \in \{x \mid x \in \mathbb{Z} \wedge x \in [-10, 15]\}$.

2.2.2.2 Uniform continuous distribution (docs)

It's the same as above, with the difference that it generates **real** numbers in the range $[a, b) \subset \mathbb{R}$.

2.2.2.3 Bernoulli distribution (docs)

Problem 3 (Network protocols)**.** To model a network protocol P it's necessary to model requests. When sent, a request can randomly fail with probability $p = 0.001$.

Generally, a random fail can be simulated by generating $r \in [0, 1]$ and checking whether $r < p$.

```
std::uniform_real_distribution<> uniform(0, 1);

if (uniform(urng) < 0.001) {
    fail();
}
```

A `std::bernoulli_distribution` is a better fit for this specification, as it generates a boolean value and its semantics represents "an event that could happen with a certain probability p".

```
std::bernoulli_distribution random_fail(0.001);

if (random_fail(urng)) {
    fail();
}
```

2.2.2.4 Normal distribution (docs)

Typical Normal distribution, requires the mean ① and the stddev ② .

```cpp
#include <iomanip>
#include <iostream>
#include <map>

#include "listings.hpp"

int main() {
    std::default_random_engine pseudo_random_engine =
        pseudo_random_engine_from_device();

    std::normal_distribution<> normal_distribution(12, 2);

    std::map<long, unsigned> histogram{};
    for (size_t _ = 0; _ < ITERATIONS; _++) {
        histogram[(size_t)normal_distribution(
            pseudo_random_engine
        )]++;
    }

    for (const auto [k, v] : histogram) {
        if (v / 200 > 0) {
            std::cout << std::setw(2) << k << ' '
                      << std::string(v / 200, '*')
                      << std::endl;
        }
    }

    return EXIT_SUCCESS;
}
```

Listing 12: `listings/normal_distribution.cpp`

```
 8 **
 9 ****
10 *******
11 *********
12 *********
13 *******
14 ****
15 **
```

2.2.2.5 Exponential distribution (docs)

Problem 4 (Cheaper servers)**.** A server receives requests at a rate of 5 requests per minute from each client. You want to rebuild the architecture of the server to make it cheaper. To test if the new architecture can handle the load, its required to build a model of client that sends requests at random intervals with an expected rate of 5 requests per minute.

It's easier to simulate the system in seconds (to have more precise measurements). If the client sends 5/min, the rate in seconds should be $\lambda = \frac{5}{60} \sim 0.083$ requests per second.

```cpp
#include <iostream>

#include "listings.hpp"

int main() {
    std::default_random_engine pseudo_random_engine =
        pseudo_random_engine_from_device();

    std::exponential_distribution<> random_time_interval(
        5.0 / 60.0
    );

    float next_request_time = 0;
    size_t total_requests = 0, total_minutes = 0;
    for (size_t time = 0; time < SIMULATION_HORIZON; time++) {
        if (time % 60 == 0) {
            total_minutes++;
        }

        if (time < next_request_time) {
            continue;
        }

        total_requests++;
        next_request_time =
            time + random_time_interval(pseudo_random_engine);
    }

    /* Should be 5. */
    std::cout << (float)total_requests / (float)total_minutes
              << std::endl;

    return EXIT_SUCCESS;
}
```

Listing 13: `listings/exponential_distribution.cpp`

The code above has a counter to measure how many requests were sent each minute. A new counter is added every 60 seconds ① , and it's incremented by 1 each time a request is sent ② . At the end, the average of the counts is calculated ③ , and it comes out to be about 5 requests every 60 seconds (or 5 requests per minute).

2.2.2.6 Poisson distribution (docs)

The Poisson distribution is closely related to the Exponential distribution, as it randomly generates a number of items in a time unit given the average rate.

```cpp
#include <iomanip>
#include <iostream>
#include <map>

#include "listings.hpp"

int main() {
    std::default_random_engine pseudo_random_engine =
        pseudo_random_engine_from_device();

    std::poisson_distribution<> poisson(4);

    std::map<long, unsigned> histogram{};
    for (size_t _ = 0; _ < ITERATIONS; _++) {
        histogram[(size_t)poisson(pseudo_random_engine)]++;
    }

    for (const auto [k, v] : histogram) {
        if (v / 100 > 0) {
            std::cout << std::setw(2) << k << ' '
                      << std::string(v / 100, '*') << '\n';
        }
    }
}
```

Listing 14: `listings/poisson_distribution.cpp`

```
 0 *
 1 *******
 2 **************
 3 *******************
 4 ********************
 5 ***************
 6 **********
 7 *****
 8 **
 9 *
```

2.2.2.7 Geometric distribution (docs)

A typical geometric distribution, has the same API as the others.

2.2.3 Discrete distribution and transition matrices (docs)

Problem 5 (E-commerce)**.** To choose the architecture for an e-commerce it's necessary to simulate realistic purchases. After interviewing 678 people it's determined that 232 of them would buy a shirt from your e-commerce, 158 would buy a hoodie and the other 288 would buy pants.

The objective is to simulate random purchases reflecting the results of the interviews. One way to do it is to calculate the percentage of buyers for each item, generate $r \in [0, 1]$, and do some checks on r. However, this specification can be implemented very easily in C++ by using a std::discrete_distribution, without having to do any calculation or write complex logic.

```cpp
#include <iostream>

#include "listings.hpp"

enum Item { Shirt = 0, Hoodie = 1, Pants = 2 };

int main() {
    std::default_random_engine pseudo_random_engine =
        pseudo_random_engine_from_device();

    std::discrete_distribution<> random_item = {232, 158, 288};

    for (size_t request = 0; request < 1000; request++) {
        switch (random_item(pseudo_random_engine)) {
        case Shirt:
            std::cout << "shirt";
            break;
        case Hoodie:
            std::cout << "hoodie";
            break;
        case Pants:
            std::cout << "pants";
            break;
        }

        std::cout << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Listing 15: listings/discrete_distribution.cpp

The rand_item instance generates a random integer $x \in \{0, 1, 2\}$ (because 3 items were sepcified in the array ① , if the items were 10, then x would have been s.t. $0 \leq x \leq 9$). The = {a, b, c} syntax can be used to intialize the a discrete distribution because C++ allows to pass a std::array to a constructor [9].

The discrete_distribution uses the in the array to generates the probability for each integer. For example, the probability to generate 0 would be calculated as $\frac{232}{232+158+288}$, the probability to generate 1 would be $\frac{158}{232+158+288}$ an the probability to generate 2 would be $\frac{288}{232+158+288}$. This way, the sum of the probabilities is always 1, and the probability is proportional to the weight.

To map the integers to the actual items ② an enum is used: for simple enums each entry can be converted automatically to its integer value (and viceversa). In C++ there is another construct, the enum class which doesn't allow implicit conversion (the conversion must be done with a function or with `static_cast`), but it's more typesafe.

The `discrete_distribution` can also be used for transition matrices, like the one in Table 1. It's enough to assign each state a number (e.g. `sunny = 0`, `rainy = 1`), and model the transition probability of **each state** as a discrete distribution.

```
std::discrete_distribution[] markov_chain_transition_matrix = {
    /* 0 */ { /* 0 */ 0.8, /* 1 */ 0.2},
    /* 1 */ { /* 0 */ 0.5, /* 1 */ 0.5}
}
```

In the example above the probability to go from state `0` `(sunny)` to `0` `(sunny)` is 0.8, the probability to go from state `0` `(sunny)` to `1` `(rainy)` is 0.2 etc...

The `discrete_distribution` can be initialized if the weights aren't already know and must be calculated.

```
for (auto &weights t1 : matrix) {
    markov_chain_transition_matrix.push_back(
        std::discrete_distribution<>(
            weights.begin(), t2  weights.end() t3 )
    );
}
```

Listing 16: `practice/2025-01-09/1/main.cpp`

The weights are stored in a `vector` ① , and the `discrete_distribution` for each state is initialized by indicating the pointer at the beginning ② and at the end ③ of the vector. This works with dynamic arrays too.

3 Code presented in lectures

Each example has 4 digits `xxxx` that are the same as the ones in the `software` folder in the course material. The code will be **as simple as possible** to better explain the core functionality, but it's **strongly suggested** to try to add structure *(classes etc..)* where it **seems fit**.

3.1 First examples

This section puts together the **formal definitions** and the `C++` knowledge to implement some simple MDPs.

3.1.1 A simple MDP `"course/1100"`

The first MDP example is $M = (U, Y, X, p, g)$ where - $U = \{\varepsilon\}$

- $Y = X$ the set of outputs matches the set of states
- $X = [0,1] \times [0,1] = [0,1]^2$ each state is a vector of two real numbers
- $p : X \times X \times U \to X$, the transition probability, is uniform over X for each input
- $g : X \to Y : x \mapsto x$ outputs the current state
- $(0,0)$ is the initial state

```cpp
#include <iostream>

#include "../software.hpp"

int main() {
    std::uniform_real_distribution<> random_uniform(0, 1);
    std::vector<real_t> MDP_state_vector(2, 0);

    const size_t HORIZON = 10;

    for (size_t time = 0; time <= HORIZON; time++) {
        std::cout << time << ' ';
        for (real_t &component : MDP_state_vector) {
            component = random_uniform(urng);
            std::cout << component << ' ';
        }
        std::cout << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Listing 17: `course/1100/main.cpp`

3.1.2 Network of MDPs pt.1 `"course/1200"`

This example has two discrete-time MDPs M_1, M_2 s.t.

- $M_1 = (U_1, Y_1, X_1, p_1, g_1)$
- $M_2 = (U_2, Y_2, X_2, p_2, g_2)$

M_1 and M_2 are similar to the MDP in Section 3.1.1 (i.e. $X = [0,1]^2$), with the difference that $\forall i \in \{1,2\}$ $U_i = X_i$, and p is redefined in this example in the following way:

$$\forall i \in \{1,2\}, x', x \in X, u \in U \quad p_i(x'|x,u) = \begin{cases} 1 \text{ if } x' = u \\ 0 \text{ otherwise} \end{cases} \tag{11}$$

Definition 4 (Discrete time steps)**.** Given a time step $t \in \mathbb{N}$, let $U(t), X(t)$ be respectively the input and state at time t.

The value of $U(t+1)$ for each MDP in this example is defined as

$$\begin{aligned} U_1(t+1) &= \begin{pmatrix} x_1 \cdot \mathcal{U}(0,1) \\ x_2 \cdot \mathcal{U}(0,1) \end{pmatrix} \text{ where } g_2(X(t)) = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\ U_2(t+1) &= \begin{pmatrix} x_1 + \mathcal{U}(0,1) \\ x_2 + \mathcal{U}(0,1) \end{pmatrix} \text{ where } g_1(X(t)) = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \end{aligned} \tag{12}$$

Thus, given that $X_i(t) \overset{1}{=} U_i(t)$ with probability 1, $g_i(X_i(t)) \overset{2}{=} X_i(t)$, and the definition in Equation 12, the connection between M_1 and M_2 can be defined as

$$\begin{aligned} X_1(t+1) &\overset{1}{=} \begin{pmatrix} x_1 \cdot \mathcal{U}(0,1) \\ x_2 \cdot \mathcal{U}(0,1) \end{pmatrix} \text{ where } X_2(t) \overset{2}{=} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\ X_2(t+1) &\overset{1}{=} \begin{pmatrix} x_1 + \mathcal{U}(0,1) \\ x_2 + \mathcal{U}(0,1) \end{pmatrix} \text{ where } X_1(t) \overset{2}{=} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \end{aligned} \tag{13}$$

With Equation 13 the code is easier to write, but this approach works only for small examples. For more complex systems it's better to design a module for each component and handle the connections more explicitly.

```
#include <iostream>

#include "../software.hpp"

struct MarkovDecisionProcess {
    real_t state_vector[2];
};

int main() {
    std::uniform_real_distribution<> random_uniform(0, 1);
    std::vector<MarkovDecisionProcess>
        markov_decision_processes(2, {0, 0});

    const size_t HORIZON = 10;

    for (size_t time = 0; time ≤ HORIZON; time++) {
        for (size_t component = 0; component < 2;
             component++) {
            markov_decision_processes[0]
                .state_vector[component] =
                markov_decision_processes[1]
```

```
                    .state_vector[component] *
                random_uniform(urng);
            markov_decision_processes[1]
                .state_vector[component] =
                markov_decision_processes[0]
                    .state_vector[component] +
                random_uniform(urng);
        }

        std::cout << time << ' ';
        for (auto markov_decision_process :
             markov_decision_processes) {
            for (auto component :
                 markov_decision_process.state_vector) {
                std::cout << component << ' ';
            }
        }
        std::cout << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Listing 18: `course/1200/main.cpp`

3.1.3 Network of MDPs pt.2 `"course/1300"`

This example is similar to the one in Section 3.1.2, with a few notable differences:

- $U_i = Y_i = X_i = \mathbb{R} \times \mathbb{R}$
- the initial states are $x_1 = (1, 1) \in X_1, x_2 = (2, 2) \in X_2$
- the connections are slightly more complex.
- no probability is involved

Having

$$p\left(\begin{pmatrix} {x_1}' \\ {x_2}' \end{pmatrix} \middle| \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}\right) = \begin{cases} 1 \text{ if } \ldots \\ 0 \text{ otherwise} \end{cases} \quad \text{where } \ldots \tag{14}$$

```
#include <iostream>

#include "../../mocc/mocc.hpp"

struct MarkovDecisionProcess {
    real_t state_vector[2];
};

int main() {
    std::vector<MarkovDecisionProcess>
        markov_decision_processes({{2, 2}, {1, 1}});
    const size_t HORIZON = 10;
    for (size_t time = 0; time <= HORIZON; time++) {
        markov_decision_processes[0].state_vector[0] =
            .7 * markov_decision_processes[0].state_vector[0] +
            .7 * markov_decision_processes[0].state_vector[1];
        markov_decision_processes[0].state_vector[1] =
            -.7 *
                markov_decision_processes[0].state_vector[0] +
            .7 * markov_decision_processes[0].state_vector[1];

        markov_decision_processes[1].state_vector[0] =
            markov_decision_processes[1].state_vector[0] +
            markov_decision_processes[1].state_vector[1];
        markov_decision_processes[1].state_vector[1] =
            -markov_decision_processes[1].state_vector[0] +
            markov_decision_processes[1].state_vector[1];

        std::cout << time << ' ';
        for (auto markov_decision_process :
             markov_decision_processes)
            for (auto component :
                 markov_decision_process.state_vector) {
                std::cout << component << ' ';
            }
        std::cout << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Listing 19: `course/1300/main.cpp`

3.1.4 Network of MDPs pt.3 `"course/1400"`

The original model behaves exactly lik Listing 19, with a different implementation. As an exercise, the reader is encouraged to come up with a different implementation for Listing 19.

3.2 Traffic light "course/2000"

This example models a traffic light. The three original versions presented in the course (2100, 2200 and 2300) all have the same behaviour, with different implementations. The code reported in this document behaves like the original versions, with a simpler implementation. Let T be the MDP that describes the traffic light where

- $U = \{\varepsilon, \sigma\}$ where
  - ε means *"do nothing"*
  - σ means *"switch light"*
- $Y = X$
- $X = \{\mathbb{G}, \mathbb{R}, \mathbb{Y}\}$ where
  - $\mathbb{G} = \text{green}$
  - $\mathbb{R} = \text{red}$
  - $\mathbb{Y} = \text{yellow}$
- $g(x) = x$

$$p(x'|x, \varepsilon) = \begin{cases} 1 & \text{if } \; x' = x \\ 0 & \text{otherwise} \end{cases}$$

$$p(x'|x, \sigma) = \begin{cases} 1 & \text{if } \; (x = \mathbb{G} \land x' = \mathbb{Y}) \lor (x = \mathbb{Y} \land x' = \mathbb{R}) \lor (x = \mathbb{R} \land x' = \mathbb{G}) \\ 0 & \text{otherwise} \end{cases}$$

Meaning that, if the input is ε, T maintains the same color with probability 1. Otherwise, if the input is σ, T changes color with probability 1 if and only if the color switch is valid (one of $\mathbb{G} \to \mathbb{Y}, \mathbb{Y} \to \mathbb{R}, \mathbb{R} \to \mathbb{G}$).

```cpp
#include <iostream>

#include "../software.hpp"

int main() {
    enum LightColor { GREEN = 0, YELLOW = 1, RED = 2 };
    const size_t HORIZON = 1000;
    std::uniform_int_distribution<> random_time_interval(
        60, 120
    );
    LightColor light = LightColor::RED;
    size_t next_switch_time = random_time_interval(urng);

    for (size_t time = 0; time ≤ HORIZON; time++) {
        std::cout << time << ' ' << next_switch_time - time
                  << ' ' << light << std::endl;
        if (time < next_switch_time) {
            continue;
        }
        light =
            (light == RED ? GREEN
                          : (light == GREEN ? YELLOW : RED));
        next_switch_time = time + random_time_interval(urng);
    }

    return EXIT_SUCCESS;
}
```

Listing 20: `course/2000/main.cpp`

To reperesent the colors the cleanest way is to use an `enum`. C++ has two types of enums: `enum` and `enum class`. In this example a simple `enum` is good enough ①, because its constants are automatically casted to their value when mapped to string ③ this doesn't happen with `enum class` because it is a stricter type, and requires explicit casting.

The behaviour of the formula described above is implemented with a couple of ternary operators ④.

3.3 Control center `"course/3100"`

This example adds complexity to the traffic light by introducing a remote control center, network faults and repairs. Having many communicating components, this example requires more structure.

The first step into building a complex system is to model it's components as units that can communicate with eachother. The traffic light needs to be to re-implemented as a component (which can be easily done with the `mocc` library).

```
#pragma once

#include "../../mocc/mocc.hpp"
#include <cstddef>
#include <random>

enum Light { GREEN = 0, YELLOW = 1, RED = 2 };
static std::random_device random_device;
static urng_t urng = urng_t(random_device());
const size_t HORIZON = 1000;
```

Listing 21: `course/3100/parameters.hpp`

The simulation requires some global variables and types in order to work, the simplest solution is to make a header file with all these data:

- `#pragma once` ① is used instead of `#ifndef xxxx #define xxxx`; it has the same behaviour (preventing multiple definitions when a file is imported multiple times)… technically `#pragma once` isn't part of the standard, yet all modern compilers support it
- `enum Light` ② has a more important job in this example: it's used to communicate values from the **controller** to the **traffic light** via the **network**; technically it could be defined in its own file, but, for the

sake of the example, it's not worth to make code navigation more complex
- there is no problem in defining global constants ③, but global variables are generally discouraged ④ (the alternative would be a singleton or passing the values as parameters to each component, but it would make the example more complex than necessary)

```cpp
#pragma once

#include "../../mocc/time.hpp"
#include "parameters.hpp"

class TrafficLight : public Observer<Timer *> {
    std::uniform_int_distribution<> random_interval;
    Light l = Light::RED;

  public:
    TrafficLight(Time &time) : random_interval(60, 120) {
        new Timer(90, TimerMode::Once, &time, this);
    }

    void update(Timer *timer) override {
        l = (l == RED ? GREEN : (l == GREEN ? YELLOW : RED));
        timer->resetWithDuration(random_interval(urng));
    }

    Light light() { return l; }
};
```

Listing 22: `course/3100/traffic_light.hpp`

By using the `mocc` library, the re-implementation of the traffic light is quite simple. A `TrafficLight` is a `Timed` component ①, which means that it has a `timer`, and whenever the `timer` reaches 0 it ⑤ it receives a notification (the method `update(U)` is called, and the traffic light switches color). The `timer` needs to be attached to a `System` for it to work ④, and must be initialized. In the library there are two types of `Timer`
- `TimerMode::Once`: when the timer ends, it doesn't automatically restart (it must be manually reset, this allows to set a different after each time the timer reaches 0, e.g. with a random variable ② ⑦)
- `TimerMode::Repeating`: the `Timer` automatically resets with the last value set

Like before, the state of the MDP is just the `Light` ③, which can be read ⑧ but not modified by external code.

```cpp
#include <fstream>

#include "parameters.hpp"
#include "traffic_light.hpp"
```

```
int main() {
    System system;
    Time time(1);
    TrafficLight traffic_light(time);

    system.addObserver(&time);

    std::ofstream file("logs");

    while (time.elapsedTime() ≤ HORIZON) {
        file << time.elapsedTime() << ' '
             << traffic_light.light() << std::endl;
        system.next();
    }

    file.close();
    return EXIT_SUCCESS;
}
```

Listing 23: `course/3100/main.cpp`

The last step is to put together the system and run it. A `System` ① is a simple MDP which sends an output ε when the `next()` method is called. By connecting all the components to the `System` it's enough to repeatedly call the `next()` method to simulate the whole system.

A `Stopwatch` ② is needed to measure how much time has passed since the simulation started, and the `TrafficLight` ③ is connected to a `timer` which itself is connected to the `System`.

3.4 Network monitor

The next objective is to introduce a control center which sends information to the traffic light via a network. The traffic light just takes the value it receives via network and displays it.

3.4.1 No faults `"course/3200"`

```
#pragma once

#include "../../mocc/time.hpp"
#include "parameters.hpp"
#include <cstdlib>

class ControlCenter : public Observer<Timer *>,
                      public Notifier<NetworkPayloadLight> {
    std::uniform_int_distribution<> random_interval;
    Light l = Light::RED;

  public:
    ControlCenter(Time &time) : random_interval(60, 120) {
        new Timer(90, TimerMode::Once, &time, this);
    }

    void update(Timer *timer) override {
        l = (l == RED ? GREEN : (l == GREEN ? YELLOW : RED));
        notify(l);
        timer->resetWithDuration(random_interval(urng));
    }

    Light light() { return l; }
};
```

Listing 24: `course/3200/control_center.hpp`

The `ControlCenter` has the same behaviour the traffic light had before, with a small difference: it notifies ① other components when the light switches. The type of the notification is `Payload` (which is just a `STRONG_ALIAS` for `Light`), this way only components that take a `Payload` (i.e. the `Network` component) can be connected to the `ControlCenter`.

```
#pragma once

#include "../../mocc/notifier.hpp"
#include "../../mocc/observer.hpp"
#include "parameters.hpp"
#include <cstdlib>

class TrafficLight : public Observer<LightUpdateMessage>,
                     public Notifier<Light> {
    Light l = Light::RED;

  public:
    void update(LightUpdateMessage light) override {
        l = light;
        notify(l);
    }

    Light light() { return l; }
};
```

Listing 25: `course/3200/traffic_light.hpp`

At this point the traffic light is easier to implement, as it just takes in input a `Message` from other components (i.e. the `Network`), changes its light ① and notifies other components ② of the change (`Message` is just a `STRONG_ALIAS` for `Light`).

```
#pragma once

#include "../../mocc/alias.hpp"
#include "../../mocc/mocc.hpp"
#include <random>
```

```
enum Light { GREEN = 0, YELLOW = 1, RED = 2 };
const size_t HORIZON = 1000;
static std::random_device random_device;
static urng_t urng = urng_t(random_device());

STRONG_ALIAS(NetworkPayloadLight, Light);
STRONG_ALIAS(LightUpdateMessage, Light);
```

Listing 26: course/3200/parameters.hpp

The STRONG_ALIASes are defined in the parameters.hpp file (it's enough to import the mocc/alias.hpp file from the library). Strong aliases are different from typedef or using aliases, as the new type is different from the type it aliases (Payload is a different type from Light), but their values can be exchanged (a Light value can be assigned to a Payload and viceversa). Aliases enable type-safe connections among components.

```
#pragma once

#include "../../mocc/buffer.hpp"
#include "../../mocc/notifier.hpp"
#include "../../mocc/time.hpp"
#include "parameters.hpp"
#include <cstdlib>

class Network : public Observer<Timer *>,
                public Buffer<NetworkPayloadLight>,
                public Notifier<LightUpdateMessage> {

  public:
    Timer *timer;

    Network(Time &time) {
        Timer *timer =
            new Timer(0, TimerMode::Once, &time, this);
    }

    void update(NetworkPayloadLight payload) override {
        if (buffer.empty()) {
            timer->resetWithDuration(2);
        }
        Buffer<NetworkPayloadLight>::update(payload);
    }

    void update(Timer *timer) override {
        if (!buffer.empty()) {
            notify((Light)buffer.front());
            buffer.pop_front();

            if (!buffer.empty()) {
                timer->resetWithDuration(2);
            }
        }
    }
};
```

Listing 27: `course/3200/network.hpp`

The simplest form of network has an illimited `Buffer` ① for the incoming messages, and every 2 seconds it sends the message to the destination (to simulate a delay). This model of the network has many problems: it doesn't account for faults (messages are corrupted / lost), buffer overflow, the fact that all messages take the same time to be sent etc...

```
#pragma once

#include "../../mocc/notifier.hpp"
#include "../../mocc/observer.hpp"
#include "parameters.hpp"
#include <cstdlib>

class TrafficLight : public Observer<LightUpdateMessage>,
                     public Notifier<Light> {
    Light l = Light::RED;

  public:
    void update(LightUpdateMessage light) override {
        l = light;
        notify(l);
    }

    Light light() { return l; }
};
```

Listing 28: `course/3200/traffic_light.hpp`

The `Monitor` is a component that takes inputs from both the `ControlCenter` and the `TrafficLight` and checks if messages are lost (a message is lost if it takes more then 3 seconds for the traffic light to change).

3.4.2 Faults & no repair `"course/3300"`

```
#pragma once

#include "../../mocc/buffer.hpp"
#include "../../mocc/notifier.hpp"
#include "../../mocc/time.hpp"
#include "parameters.hpp"
#include <cstdlib>
#include <random>

class Network : public Observer<Timer *>,
                public Buffer<NetworkPayloadLight>,
                public Notifier<LightUpdateMessage> {
    std::bernoulli_distribution random_fault;

  public:
    Timer *timer;

    Network(Time &time) : random_fault(0.05) {
        timer = new Timer(0, TimerMode::Once, &time, this);
    }

    void update(NetworkPayloadLight payload) override {
        if (buffer.empty()) {
            timer->resetWithDuration(2);
        }
        Buffer<NetworkPayloadLight>::update(payload);
    }

    void update(Timer *timer) override {
        if (!buffer.empty()) {
            if (!random_fault(urng)) {
                notify((Light)buffer.front());
            }
            buffer.pop_front();
            if (!buffer.empty()) {
                timer->resetWithDuration(2);
            }
        }
    }
};
```

Listing 29: `course/3300/network.hpp`

The first change is to add faults to the network ①, which can be done easily by using a `std::bernoulli_distribution` with a certain fault probability (e.g. 0.01), and send the message only if there is no fault. Once the message is lost nothing can be done, the system doesn't recover.

3.4.3 Faults & repair "course/3400"

```
#pragma once

#include "../../mocc/buffer.hpp"
#include "../../mocc/notifier.hpp"
#include "../../mocc/time.hpp"
#include "parameters.hpp"
#include <cstdlib>
#include <random>

class Network : public Observer<Timer *>,
                public Buffer<NetworkPayloadLight>,
                public Notifier<LightUpdateMessage> {

    std::bernoulli_distribution random_fault;
    std::bernoulli_distribution random_repair;

  public:
    Timer *timer;

    Network(Time &time)
        : random_fault(0.01), random_repair(0.001) {
        timer = new Timer(0, TimerMode::Once, &time, this);
    }

    void update(NetworkPayloadLight payload) override {
        if (buffer.empty()) {
            timer->resetWithDuration(2);
        }
        Buffer<NetworkPayloadLight>::update(payload);
    }

    void update(Timer *timer) override {
        if (!buffer.empty()) {
            if (!random_fault(urng) || random_repair(urng)) {
                notify((Light)buffer.front());
            }
            buffer.pop_front();
            if (!buffer.empty()) {
                timer->resetWithDuration(2);
            }
        }
    }
};
```

Listing 30: course/3400/network.hpp

The next idea is to add repairs ① when the system fails. In this case the repairs are random for simplicity ②, but there are smarter ways to handle a network fault.

3.4.4 Faults & repair + protocol "course/3500"

```
#pragma once

#include "../../mocc/buffer.hpp"
#include "../../mocc/notifier.hpp"
#include "../../mocc/time.hpp"
#include "parameters.hpp"
#include <cstdlib>
#include <random>

class Network : public Observer<Timer *>,
                public Buffer<NetworkPayloadLight>,
                public Notifier<LightUpdateMessage>,
                public Notifier<Fault> {
    std::bernoulli_distribution random_fault;
    std::bernoulli_distribution random_repair;

  public:
    Timer *timer;

    Network(Time &time)
        : random_fault(0.01), random_repair(0.001) {
        timer = new Timer(0, TimerMode::Once, &time, this);
    }

    void update(NetworkPayloadLight payload) override {
        if (buffer.empty()) {
            timer->resetWithDuration(2);
        }
        Buffer<NetworkPayloadLight>::update(payload);
    }

    void update(Timer *timer) override {
        if (!buffer.empty()) {
            if (random_fault(urng)) {
                if (random_repair(urng)) {
                    Notifier<LightUpdateMessage>::notify(
                        (Light)buffer.front()
                    );
                } else {
                    Notifier<Fault>::notify(true);
                }
            } else {
                Notifier<LightUpdateMessage>::notify(
                    (Light)buffer.front()
                );
            }

            buffer.pop_front();
            if (!buffer.empty()) {
                timer->resetWithDuration(2);
            }
        }
    }
};
```

Listing 31: course/3500/network.hpp

In the last version, the network sends a notification ② when there is a Fault ① (which is just a STRONG_ALIAS for bool), this way the TrafficLight can recover in case of errors.

```
#pragma once

#include "../../mocc/notifier.hpp"
#include "../../mocc/observer.hpp"
#include "parameters.hpp"
#include <cstdlib>

class TrafficLight : public Observer<Fault>,
                     public Observer<LightUpdateMessage>,
                     public Notifier<Light> {
    Light l = Light::RED;

  public:
    void update(LightUpdateMessage light) override {
        l = light;
        notify(l);
    }

    void update(Fault) override {
        l = Light::RED;
        notify(l);
    }

    Light light() { return l; }
};
```

Listing 32: course/3500/traffic_light.hpp

When the TrafficLight detects a Fault it turns to Light::RED for safety reasons.

```
#include <cstdlib>
#include <fstream>

#include "../../mocc/notifier.hpp"
#include "../../mocc/system.hpp"
#include "control_center.hpp"
#include "monitor.hpp"
#include "network.hpp"
#include "parameters.hpp"
#include "traffic_light.hpp"

int main() {
    System system;
    Monitor monitor;
    Time time(1, &system);
    Network network(time);
    TrafficLight traffic_light;
    ControlCenter control_center(time);

    system.addObserver(&monitor);
    network.Notifier<LightUpdateMessage>::addObserver(
        &traffic_light
    );
    network.Notifier<Fault>::addObserver(&traffic_light);
    traffic_light.addObserver(&monitor);
    control_center.addObserver(&network);
    control_center.addObserver(&monitor);

    std::ofstream file("logs");

    while (time.elapsedTime() <= HORIZON) {
        file << time.elapsedTime() << ' '
             << control_center.light() << ' '
             << traffic_light.light() << ' '
             << monitor.isValid() << std::endl;

        system.next();
    }

    file.close();
    return EXIT_SUCCESS;
}
```

Listing 33: `course/3500/main.cpp`

3.5 Statistics

3.5.1 Expected value `"/course/4100"`

In this example the goal is to simulate a development process (phase 0, phase 1, and phase 2), and calculate the cost of each simulation.

```
#include <cstddef>
#include <fstream>
#include <random>

#include "../../mocc/mocc.hpp"

int main() {
    std::random_device random_device;
    urng_t urng(random_device());

    // clang-format off
    std::vector<std::discrete_distribution<>>
        transition_matrix = {
            {0, 1},
            {0, .3, .7},
            {0, 0, .2, .8},
            {0, .1, .1, .1, .7},
            {0, 0, 0, 0, 1},
        };
    // clang-format on

    real_t time = 0;
    size_t phase = 0, costs = 0;

    std::ofstream file("logs");

    while (phase ≠ 4) {
        time++;
        if (phase == 1 || phase == 3)
            costs += 20;
        else if (phase == 2)
            costs += 40;

        phase = transition_matrix[phase](urng);
        file << time << ' ' << phase << ' ' << costs
             << std::endl;
    }

    file.close();
    return EXIT_SUCCESS;
}
```

Listing 34: `course/4100/main.cpp`

3.5.2 Probability `"/course/4200"`

This example behaves like the previous one, but uses the Monte Carlo method [3] to calculate the probability the cost is less than a certain value

```
#include <cstddef>
#include <fstream>
#include <iostream>
#include <random>

#include "../../mocc/math.hpp"
#include "../../mocc/mocc.hpp"

const size_t ITERATIONS = 10000;

int main() {
    std::random_device random_device;
    urng_t urng(random_device());

    // clang-format off
    std::vector<std::discrete_distribution<>>
        transition_matrix = {
            {0, 1},
            {0, .3, .7},
            {0, 0, .2, .8},
            {0, .1, .1, .1, .7},
            {0, 0, 0, 0, 1},
        };
    // clang-format on

    DataDistribution costs_distribution;
    size_t less_than_100_count = 0;
    real_t time = 0;

    std::ofstream file("logs");

    for (int iter = 0; iter < ITERATIONS; iter++) {
        size_t phase = 0, costs = 0;

        while (phase != 4) {
            time++;
            if (phase == 1 || phase == 3)
                costs += 20;
            else if (phase == 2)
                costs += 40;

            phase = transition_matrix[phase](urng);
            file << time << ' ' << phase << ' ' << costs
                 << std::endl;
        }

        costs_distribution.insertDataPoint(costs);
        if (costs < 100)
            less_than_100_count++;
    }

    std::cout << costs_distribution.mean() << ' '
              << costs_distribution.stddev() << ' '
              << (double)less_than_100_count / ITERATIONS
              << std::endl;

    file.close();
    return EXIT_SUCCESS;
}
```

Listing 35: `course/4200/main.cpp`

3.6 Development process simulation

An MDP can be implemented by using a **transition matrix** The simplest implemenation can be done by using a `std::discrete_distribution` by using the trick in Listing 10.

3.6.1 Random transition matrix `"course/5100"`

This example builds a **random transition matrix**.

```cpp
#include <fstream>
#include <random>
#include <vector>

#include "../../mocc/mocc.hpp"

const size_t HORIZON = 20, STATES_SIZE = 10;

int main() {
    urng_t urng = pseudo_random_engine_from_device();

    auto random_state =
        std::uniform_int_distribution<>(0, STATES_SIZE - 1);
    std::uniform_real_distribution<> uniform(0, 1);

    std::vector<std::discrete_distribution<>>
        transition_matrix(STATES_SIZE);

    for (size_t state = 0; state < STATES_SIZE; state++) {
        std::vector<real_t> weights(STATES_SIZE);
        for (auto &weight : weights) {
            weight = uniform(urng);
        }

        transition_matrix[state] =
            std::discrete_distribution<>(
                weights.begin(), weights.end()
            );
    }

    std::ofstream file("logs");

    size_t state = random_state(urng);
    for (size_t time = 0; time <= HORIZON; time++) {
        file << time << " " << state << std::endl;
        state = transition_matrix[state](urng);
    }

    file.close();
    return EXIT_SUCCESS;
}
```

Listing 36: `course/5100/main.cpp`

A **transition matrix** is a `vector<discrete_distribution<>>` ② just like in Listing 10. Why can we do this? First of all, the states are numbered from `0` to `STATES_SIZE - 1`, that's why we can generate a random state ① just by generating a number from `0` to `STATES_SIZE - 1`.

The problem with using a simple `uniform_int_distribution` is that we don't want to choose the next state uniformly, we want to do something like in Figure 5.

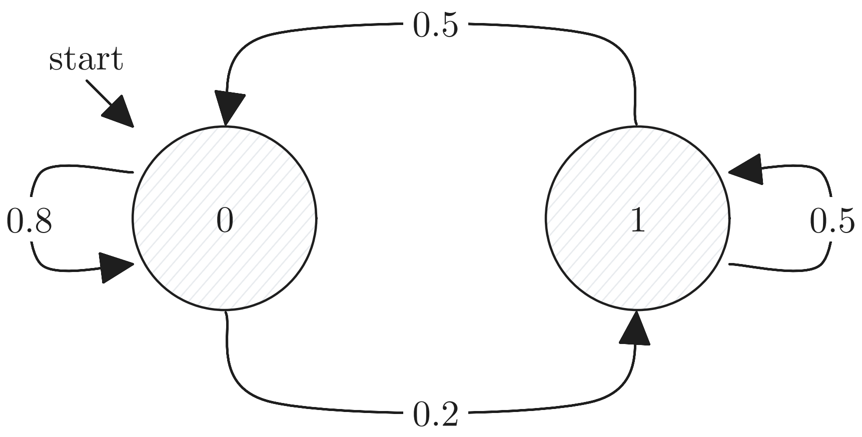

Figure 5: A simple Markov Chain

Luckly for us `std::discrete_distribution<>` does exactly what we want. It takes a list of weights $w_0, w_1, w_2, ..., w_n$ and assigns each index i the probability $p(i) = \frac{\sum_{i=0}^{n} w_i}{w_i}$ (the probability is proportional to the weight, so we have that $\sum_{i=0}^{n} p(i) = 1$ like we would expect in a Markov Chain).

To instantiate the `discrete_distribution` ④, unlike in Listing 10, we need to first calculate the weights ③, as we don't know them in advance.

To randomly generate the next state ⑥ we just have to use the `discrete_distribution` assigned to the current state ⑤.

3.6.2 Software development & error detection `"course/5200"`

Our next goal is to model the software development process of a team. Each phase takes the team 4 days to complete, and, at the end of each phase the testing team tests the software, and there can be 3 outcomes:

- **no error** is introduced during the phase (we can't actually know it, let's suppose there is an all-knowing "oracle" that can tell us there aren't any errors)
- **no error detected** means that the "oracle" detected an error, but the testing team wasn't able to find it
- **error detected** means that the "oracle" detected an error, and the testing team was able to find it

If we have **no error**, we proceed to the next phase… the same happens if **no error was detected** (because the testing team sucks and didn't find any errors). If we **detect an error** we either reiterate the current phase (with a certain probability, let's suppose 0.8), or we go back to one of the previous phases with equal probability (we do this because, if we find an error, there's a high chance it was introduced in the current phase, and we want to keep the model simple).

In this exercise we take the parameters for each phase (the probability to introduce an error and the probability to not detect an error) from a file.

```
#include <cassert>
#include <fstream>
#include <iostream>
#include <random>
#include <vector>

typedef double real_t;

#define HORIZON 800
#define PHASES_SIZE 3
#define ITERATIONS 1000

/* Se in una fase viene introddotto in errore, potrebbe essere
 * non essere rilevato. */
enum Outcome {
    NO_ERROR = 0,
    NO_ERROR_DETECTED = 1,
    ERROR_DETECTED = 2
};

int main() {
    std::random_device random_device;
    std::default_random_engine urng(random_device());

    std::uniform_real_distribution<> uniform_0_1(0, 1);
    std::vector<std::discrete_distribution<>>
        phases_error_distribution;

    {
        std::ifstream probabilities("probabilities.csv");
        real_t probability_error_introduced,
            probability_error_not_detected;

        while (probabilities >> probability_error_introduced >>
               probability_error_not_detected)
            /* vedere l'enum `Error` per capire il ragionamento
             */
            phases_error_distribution.push_back(
                std::discrete_distribution<>({
                    1 - probability_error_introduced,
                    probability_error_introduced *
                        probability_error_not_detected,
                    probability_error_introduced *
                        (1 - probability_error_not_detected),
                })
            );

        probabilities.close();
        assert(
            phases_error_distribution.size() == PHASES_SIZE
        );
    }

    std::ofstream log("log.csv");
    log << "time phase progress-0 progress-1 progress-2 "
           "outcome-0 outcome-1 "
           "outcome-2 assess-0 assess-1 assess-2"
        << std::endl;
    /* se viene rilevato un errore è molto probabile dover
     * ripetere la fase */
    real_t probability_repeat_phase = 0.8;

    size_t phase = 0;
    std::vector<size_t> progress(PHASES_SIZE, 0);
    std::vector<Outcome> outcomes(PHASES_SIZE, NO_ERROR);

    for (size_t time = 0; time < HORIZON; time++) {
        progress[phase]++;

        /* una fase dura 4 giorni esatti */
        if (progress[phase] == 4) {
            /* al termine della fase si rientra in uno dei
             * possibili casi */
            outcomes[phase] = static_cast<Outcome>(
                phases_error_distribution[phase](urng)
            );
            switch (outcomes[phase]) {
            case NO_ERROR:
            case NO_ERROR_DETECTED:
                phase++;
                break;
            case ERROR_DETECTED:
                if (phase > 0 && uniform_0_1(urng) >
                                     probability_repeat_phase)
                    phase = std::
                        uniform_int_distribution<>(0, phase - 1)(
                            urng
                        );
                break;
            }

            if (phase == PHASES_SIZE)
                break;

            progress[phase] = 0;
        }

        {
            log << time << " " << phase << " ";
            for (size_t phase = 0; phase < PHASES_SIZE;
                 phase++)
                log << progress[phase] << " ";
            for (size_t phase = 0; phase < PHASES_SIZE;
                 phase++)
                log << (outcomes[phase] == NO_ERROR ? 0 : 1)
                    << " ";
            for (size_t phase = 0; phase < PHASES_SIZE;
                 phase++)
                log << (outcomes[phase] == ERROR_DETECTED)
                    << " ";
            log << std::endl;
        }
    }

    log.close();
    return EXIT_SUCCESS;
}
```

Listing 37: `course/5300/main.cpp`

TODO: `class enum` vs `enum`. We can model the outcomes as an `enum` ①... we can use the `discrete_distribution` trick to choose randomly one of the outcomes ②. The other thing we notice is that we take the probabilities to generate an error and to detect it from a file.

3.6.3 Optimizing development costs `"course/5300"`

If we want we can manipulate the “parameters” in real life: a better experienced team has a lower probability to introduce an error, but a higher cost. What we can do is:

1. randomly generate the parameters (probability to introduce an error and to not detect it)
2. simulate the development process with the random parameters

By repeating this a bunch of times, we can find out which parameters have the best results, a.k.a generate the lowest development times (there are better techniques like simulated annealing, but this one is simple enough for us).

3.6.4 Key performance index `"course/5400"`

We can repeat the process in exercise `[5300]`, but this time we can assign a parameter a certain cost, and see which parameters optimize cost and time (or something like that? Idk, I should look up the code again).

3.7 Complex systems

3.7.1 Insulin pump `"course/6100"`

3.7.2 Buffer `"course/6200"`

3.7.3 Server `"course/6300"`

4 Exam

In short, every system can be divided into 4 steps:

- reading parameters from a file (from files as of 2024/2025)
- initializing the system
  - this include instantiating the MDPs and connecting them
- simulating the system
- saving outputs to a file

```cpp
std::ifstream params("parameters.txt");
char c;

while (params >> c) t1
    switch (c) {
        case 'A': params >> A; break;
        case 'B': params >> B; break;
        case 'C': params >> C; break;
        case 'D': params >> D; break;
        case 'F': params >> F; break;
        case 'G': params >> G; break;
        case 'N': params >> N; break;
        case 'W': params >> W; break;
    }

params.close();
```

Listing 38: `practice/1/main.cpp`

Reading the input: `std::ifstream` can read (from a file) based on the type of the variable read. For exapmle, `c` is a `char`, so ① will read exactly 1 character. If `c` was a string, `params >> c` would have read a whole word (up to the first whitespace). For example, `A` is a float and `N` is a int, so `params >> A` will try to read a float and `params >> N` will **try** to read an int. (TODO: float → real_t, int → size_t)

```cpp
#ifndef PARAMETERS_HPP_
#define PARAMETERS_HPP_

#include "../../mocc/alias.hpp" t1
#include "../../mocc/mocc.hpp" t2

STRONG_ALIAS(ProjInit, real_t) t3
STRONG_ALIAS(TaskDone, real_t) t3
STRONG_ALIAS(EmplCost, real_t) t3

static t4 real_t A, B, C, D, F, G;
static size_t N, W, HORIZON = 100000;

#endif
```

Listing 39: `practice/1/parameters.hpp`

The parameters are declared in a `parameters.hpp` file, for a few reasons
- they are declared globally, and are globally accessible without having to pass to classes constructors
- any class can just import the file with the parameters to access the parameters
- they are static ④ (otherwise clang doesn't like global variables)
- in `parameters.hpp` there are also auxiliary types ③, used in the connections between entities

```
System system; t1
Stopwatch stopwatch; t2

system.addObserver(&stopwatch); t3

while (stopwatch.elapsed() <= HORIZON) t4
    system.next(); t5
```

Simulating the system is actually easy:
- declare the system ①
- add a stopwatch ② (which starts from time 0, and everytime the system is updated, it adds up time)
  - it is needed to stop the simulation after a certain amount of time, called `HORIZON`
- connect the stopwatch to the system ③
- run a loop (like how a game loop would work) ④
- in the loop, transition the system to the next state ⑤

4.1 Development team (time & cost)

4.1.1 Employee

```
#ifndef EMPLOYEE_HPP_
#define EMPLOYEE_HPP_

#include <random>

#include "../../mocc/stat.hpp"
#include "../../mocc/time.hpp"
#include "parameters.hpp"

class Employee : public Observer<T>,
                 public Observer<ProjInit>,
                 public Notifier<TaskDone, EmplCost> {

    std::vector<std::discrete_distribution<>>
        transition_matrix;
    urng_t &urng;
    size_t phase = 0;
    real_t proj_init = 0;
```

```
 public:
  const size_t id;
  const real_t cost;
  Stat comp_time_stat;

  Employee(urng_t &urng, size_t k)
    : urng(urng), id(k),
     cost(1000.0 - 500.0 * (real_t)(k - 1) / (W - 1)) {

    transition_matrix =
      std::vector<std::discrete_distribution<>>(N);

    for (size_t i = 1; i < N; i++) {
      size_t i_0 = i - 1;
      real_t tau = A + B * k * k + C * i * i + D * k * i,
          alpha = 1 / (F * (G * W - k));

      std::vector<real_t> p(N, 0.0);
      p[i_0] = 1 - 1 / tau;
      p[i_0 + 1] =
        (i_0 == 0 ? (1 - p[i_0])
              : (1 - alpha) * (1 - p[i_0]));

      for (size_t prev = 0; prev < i_0; prev++)
        p[prev] = alpha * (1 - p[i_0]) / i_0;

      transition_matrix[i_0] =
        std::discrete_distribution<>(p.begin(),
                                     p.end());
    }

    transition_matrix[N - 1] =
      std::discrete_distribution<>{1};
  }

  void update(T t) override {
    if (phase < N - 1) {
      phase = transition_matrix[phase](urng);
      if (phase == N - 1) {
        comp_time_stat.save(t - proj_init);
        notify((real_t)t, cost);
      }
    }
  };

  void update(ProjInit proj_init) override {
    this->proj_init = proj_init;
    phase = 0;
  };
};

#endif
```

Listing 40: practice/1/employee.hpp

Bibliography

[1] Marcin Kolny, “Scaling up the Prime Video Audio-Video Monitoring Service and Reducing Costs by 90%.” Accessed: Mar. 25, 2024. [Online]. Available: https://web.archive.org/web/20240325042615/https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90#expand

[2] [Online]. Available: https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_online_algorithm

[3] Wikipedia, “Monte Carlo method.” [Online]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_method

[4] “Embarrassingly parallel.” [Online]. Available: https://en.wikipedia.org/wiki/Embarrassingly_parallel

[5] “operator — Standard operators as functions.” [Online]. Available: https://docs.python.org/3/library/operator.html

[6] “Module ops.” [Online]. Available: https://doc.rust-lang.org/std/ops/index.html#examples

[7] “std::basic_ostream.” [Online]. Available: https://en.cppreference.com/w/cpp/io/basic_ostream/operator_ltlt

[8] “Pseudo-random number generation.” [Online]. Available: https://en.cppreference.com/w/cpp/numeric/random

[9] [Online]. Available: https://en.cppreference.com/w/cpp/container/array