Software Engineering

Cicio Ionuț

27/02/2025

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 Software models

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.

1.1 The *"Amazon Prime Video"* article

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

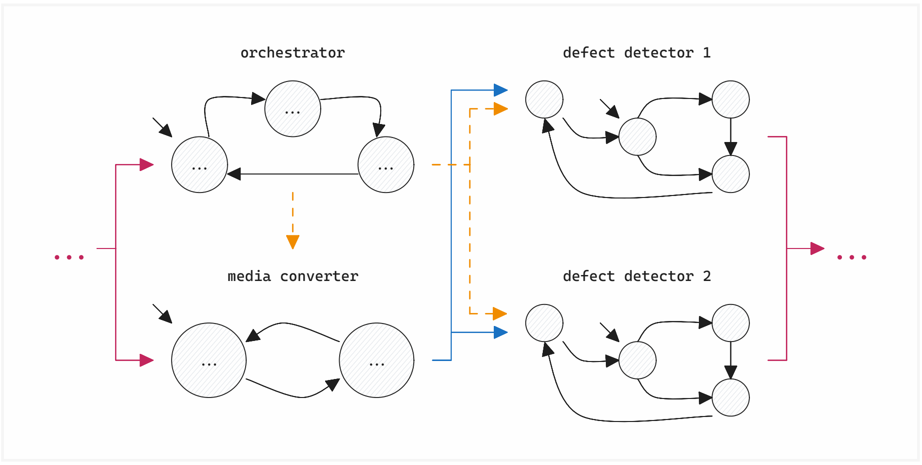

Figure 1: audio/video monitoring system process

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

1.2 Models

1.2.1 Markov chain

In simple terms, a Markov chain M is described by a set of **states** S and the **transition probability** $p : S \times S \to [0,1]$ such that $p(s'|s)$ is the probability to transition from s to s'. The transition probability p is constrained by 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 (called the Markov property), meaning that predictions can be made solely on its present state, and aren't influenced by its history.

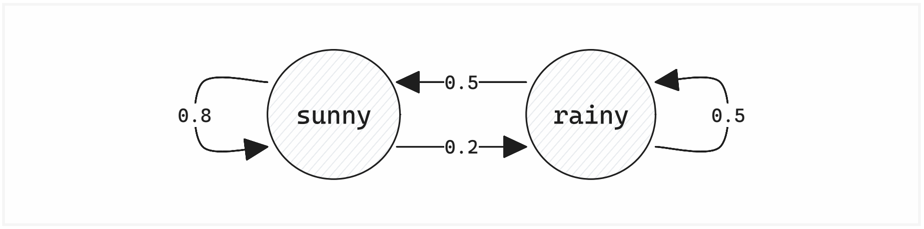

Figure 2: Example Markov chain with $S = \{\text{rainy}, \text{sunny}\}$

	sunny	rainy
sunny	0.8	0.2
rainy	0.5	0.5

Table 1: Transition probability of Figure 2

If a Markov chain M transitions at **discrete time** steps (i.e. the time steps $t_0, t_1, t_2, \ldots$ 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.

The Markov process is characterized by a **transition matrix** which describes the probability of certain transitions, like the on in Table 1. Later in the guide it will be shown that implementing transition matrices in `C++` is really simple when using the `<random>` library.

1.2.2 Markov decision process

A Markov decision process (MDP), despite sharing the name, is **different** from a Markov chain, because it interacts with an **external environment**. A MDP M is a tuple (U, X, Y, p, g) s.t.

- U is the set of **input values**
- X is the set of **states**
- Y is the set of **output values**

- $p : X \times X \times U \to [0,1]$ is such that $p(x'|x,u)$ is the probability to **transition** from state x to state x' when the **input value** is u
- $g : X \to Y$ is the **output function**
- $x_0 \in X$ is the **initial state**

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.

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

1.2.2.1 MDP example

The development process of a company can be modeled as a MDP $M = (U, X, Y, p, g)$ s.t.

- $U = \{\varepsilon\}$[1], $X = \{0,1,2,3,4\}, Y = \text{Cost} \times \text{Duration}, x_0 = 0$

$$g(x) = \begin{cases} (0,0) & x \in \{0,4\} \\ (20000,2) & x \in \{1,3\} \\ (40000,4) & x = 2 \end{cases} \tag{3}$$

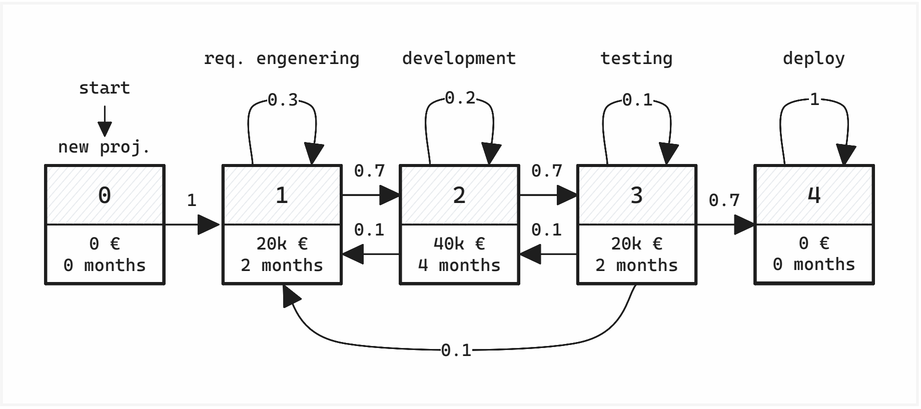

Figure 3: the model of a team's development process

ε	**0**	**1**	**2**	**3**	**4**
0	0	1	0	0	0
1	0	.3	.7	0	0
2	0	.1	.2	.7	0
3	0	.1	.1	.1	.7
4	0	0	0	0	1

Only **1 transition matrix** is needed, as $|U| = 1$ (there's 1 input value). If U had multiple input values, like $\{\text{on}, \text{off}, \text{wait}\}$, then 3 transition matrices would have been required, one **for each input value**.

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

1.2.3 Network of MDPs

Let M_1, M_2 be two MDPs s.t.

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

Then $M = (U_1, X_1 \times X_2, Y_2, p, g)$ s.t.

- $p((x_{1'}, x_{2'}) \mid (x_1, x_2), u_1) = (p(x_{1'}|x_1, u_1), p(x_{2'}|x_2, g_1(x_1)))$
- $g((x_1, x_2)) = g_2(x_2)$
- TODO the connection can also be partial

1.3 Other methods

1.3.1 Incremental average

Given a set of values $X = \{x_1, ..., x_n\} \subset \mathbb{R}$ the average $\overline{x}_n = \frac{\sum_{i=0}^{n} x_i}{n}$ can be computed with a simple procedure

```
float average(std::vector<float> X) {
    float sum = 0;
    for (auto x_i : X)
        sum += x_i;

    return sum / X.size();
}
```

Listing 1: `examples/average.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.

$$\begin{aligned}\overline{x}_{n+1} = \frac{\sum_{i=0}^{n+1} x_i}{n+1} = \frac{\left(\sum_{i=0}^{n} x_i\right) + x_{n+1}}{n+1} = \frac{\sum_{i=0}^{n} x_i}{n+1} + \frac{x_{n+1}}{n+1} = \\ \frac{\left(\sum_{i=0}^{n} x_i\right) n}{(n+1)n} + \frac{x_{n+1}}{n+1} = \frac{\sum_{i=0}^{n} x_i}{n} \cdot \frac{n}{n+1} + \frac{x_{n+1}}{n+1} = \\ \overline{x}_n \cdot \frac{n}{n+1} + \frac{x_{n+1}}{n+1}\end{aligned} \tag{4}$$

With this formula the numbers added up are smaller: $\overline{x}_n$ is multiplied by $\frac{n}{n+1} \sim 1$, and, if x_{n+1} fits in IEEE-754, then $\frac{x_{n+1}}{n+1}$ can also be encoded.

```
float incr_average(std::vector<float> X) {
    float average = 0;
    for (size_t n = 0; n < X.size(); n++)
        average =
            average * ((float)n / (n + 1)) + X[n] / (n + 1);

    return average;
}
```

Listing 2: `examples/average.cpp`

In `examples/average.cpp` the procedure `average()` returns `Inf` and `incr_average()` successfuly computes the average.

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.

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

Given M_2, if $n > 0$, the standard deviation is $\sqrt{\frac{M_{2,n}}{n}}$ ③ . The average can be calculated incrementally like in Section 1.3.1 ① .

```
void Stat::save(real_t x_i) {
    real_t next_mean =
        mean_ * ((real_t)n / (n + 1)) + x_i / (n + 1); ①

    m_2__ += (x_i - mean_) * (x_i - next_mean); ②
    mean_ = next_mean;
    n++;
}

real_t Stat::stddev() const {
    return n > 0 ? sqrt(m_2__ / n) : 0; ③
}
```

Listing 3: `mocc/math.cpp`

1.3.3 Euler method

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:

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

Let the function y be the solution to the following problem

$$\begin{cases} y(x_0) = y_0 \\ y'(x) = f(x, y(x)) \end{cases} \tag{7}$$

Let $y(x_0) = y_0$ be the initial condition of the system, and $y' = f(x, y(x))$ be the known **derivative** of y (y' is a function of x and $y(x)$). To approximate y, a Δ is chosen (the smaller, the more precise the approximation) s.t. $x_{n+1} = x_n + \Delta$. Now, understanding Equation 6 should be easier: the value of y at the **next step** is the current value of y plus the value of its derivative y' (multiplied by Δ). In Equation 6 y' is multiplied by Δ because when going to the next step, all the derivatives from x_n to x_{n+1} must be added up, and it's done by adding up

$$(x_{n+1} - x_n) \cdot f(x_n, y_n) = \Delta \cdot f(x_n, y_n) \tag{8}$$

Where $y_n = y(x_n)$. Let's consider the example in Equation 9.

$$\begin{cases} y(x_0) = 0 \\ y'(x) = 2x \end{cases}, \quad \text{with } \Delta = 1, 0.5, 0.3, 0.25 \tag{9}$$

The following program approximates Equation 9 with different Δ values.

```cpp
#define SIZE 4
float derivative(float x) { return 2 * x; }

int main() {
    size_t x[SIZE];
    for (size_t i = 0; i < SIZE; i++) {
        x[i] = 0;
        float delta = 1. / (i + 1);
        for (float t = 0; t ≤ 10; t += delta)
            x[i] += delta * derivative(t);
    }
    return 0;
}
```

Listing 4: `examples/euler.cpp`

The approximation is close, but not very precise. However, the error analysis is beyond this guide's scope.

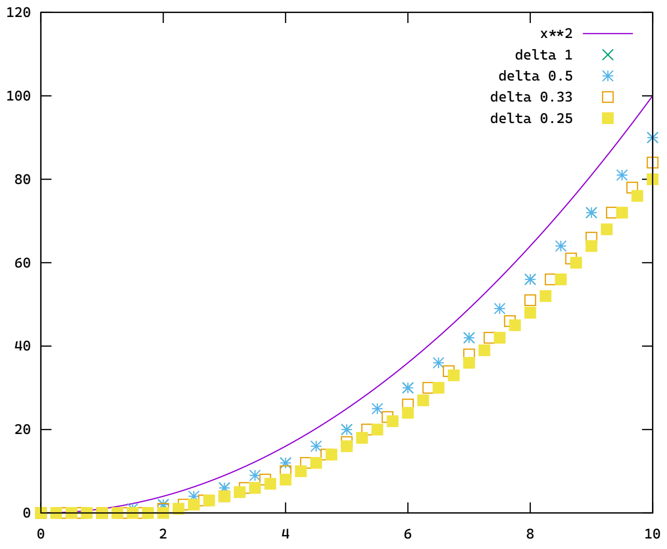

Figure 4: examples/euler.svg

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.” [2]

“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” [2]

The cost to develop a feature is described by an uniform discrete distribution $\mathcal{U}\{300, 1000\}$. Determine the probability that the cost is less than 550.

The problem above can be easily solved analitically, but let’s use the Monte Carlo method to approximate its value.

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

int main() {
    std::random_device random_device;
    std::default_random_engine rand_engine(random_device());
    std::uniform_int_distribution<> rand_cost(300, 1000);

    const size_t ITERATIONS = 10000;
    size_t below_550 = 0;

    for (size_t i = 0; i < ITERATIONS; i++) ①
        if (rand_cost(rand_engine) < 550) ②
            below_550++; ③

    std::cout << (double)below_550/ITERATIONS ④ <<
std::endl;
    return 0;
}
```

The first step is to simulate for a certain number of iterations ① the system (in this example, “simulating the system” means generating a random integer cost between 300 and 1000 ②). If the the iteration respects the requested condition, then it’s counted ③.

At the end of the simulations, the probability is calculated as $\frac{\text{iterations below 550}}{\text{total iterations}}$ ④ . The bigger is the number of iterations, the more precise is the approximation. This type of calculation can be very easily distributed in a HPC (high performance computing) context.

2 How to `C++`

This section covers the basics assuming the reader already knows `C`.

2.1 The `random` library

The `C++` standard library offers tools to easily implement MDPs.

2.1.1 Random engines

In `C++` there are many ways to **generate random numbers** [3]. Generally it's not recommended to use `random()` ① . It's better to use a **random generator** ⑤, 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() ① << std::endl;

    std::random_device random_device; ②
    std::cout << random_device() ③ << std::endl;

    int seed = random_device(); ④
    std::default_random_engine random_engine(seed); ⑤
    std::cout << random_engine() ⑥ << std::endl;
}
```

Listing 6: `examples/random.cpp`

The typical course of action is to instantiate a `random_device` ②, and use it to generate a seed ④ for a `random_engine`. Given that random engines can be used with distributions, they're really useful to implement MDPs. Also, ③ and ⑥ are examples of **operator overloading** (Section 2.4).

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

```cpp
#include <random>
// works like typedef in C
using urng_t = std::default_random_engine;

int main() {
    urng_t urng(190201);
}
```

2.1.2 Distributions

Just the capability to generate random numbers isn't enough, these numbers need to be manipulated to fit certain needs. Luckly, `C++` covers **basically all of them**. For example, the MDP in Figure 3 can be easily simulated with the following code code:

```cpp
#include <iostream>
#include <random>
using urng_t = std::default_random_engine;

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

    std::discrete_distribution<> transition_matrix[] = {
        {0, 1},
        {0, .3, .7},
        {0, .1, .2, .7},
        {0, .1, .1, .1, .7},
        {0, 0, 0, 0, 1},
    };

    size_t state = 0;
    for (size_t step = 0; step < 15; step++) {
        state = transition_matrix[state](urng);
        std::cout << state << std::endl;
    }

    return 0;
}
```

Listing 8: `examples/transition_matrix.cpp`

2.1.2.1 Uniform discrete distribution (docs)

> To test a system S it's requried to build a generator that sends value v_t to S every T_t seconds. For each send, the value of T_t is an **integer** chosen uniformly in the range $[20, 30]$.

The `C` code to compute T_t would be `T = 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:

```cpp
std::uniform_int_distribution<> random_T(20, 30); ①
size_t T = ② random_T(urng);
```

The interval T_t can be easily generated ② without needing to remember any formula or trick. The behaviour of T_t is defined only once ①, so it can be easily changed without introducing bugs or inconsistencies. It's also worth to take a look at the implementation of the exercise above (with the addition that $v_t = T_t$), as it comes up very often in software models.

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

using urng_t = std::default_random_engine;

int main() {
    std::random_device random_device;
    urng_t urng(random_device());
    std::uniform_int_distribution<> random_T(20, 30);

    size_t T = random_T(urng), next_request_time = T;
    for (size_t time = 0; time < 1000; time++) {
        if (time < next_request_time)
            continue;

        std::cout << T << std::endl;
        T = random_T(urng);
        next_request_time = time + T;
    }

    return 0;
}
```

Listing 9: `examples/interval_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

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

`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 $z \in \{x \mid x \in \mathbb{Z} \land x \in [-10, 15]\}$.

2.1.2.2 Uniform continuous distribution (docs)

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

2.1.2.3 Bernoulli distribution (docs)

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();
```

2.1.2.4 Normal distribution (docs)

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

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

using urng_t = std::default_random_engine;

int main() {
    std::random_device random_device;
    urng_t urng(random_device());
    std::normal_distribution<> normal(12 ①, 2 ②);

    std::map<long, unsigned> histogram{};
    for (size_t i = 0; i < 10000; i++)
        ++histogram[(size_t)normal(urng)];

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

    return 0;
}
```

Listing 10: `examples/normal.cpp`

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

2.1.2.5 Exponential distribution (`docs`)

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
int main() {
    std::random_device random_device;
    urng_t urng(random_device());
    std::exponential_distribution<> random_interval(5. /
60.);

    real_t next_request_time = 0;
    std::vector<real_t> req_per_min = {0};
    for (real_t time_s = 0; time_s < HORIZON; time_s++) {
        if (((size_t)time_s) % 60 == 0)
            req_per_min.push_back(0); ①

        if (time_s < next_request_time)
            continue;

        req_per_min.back()++; ②
        next_request_time = time_s + random_interval(urng);
    }

    real_t mean = 0;
    for (auto x : req_per_min) ③
        mean += x;

    std::cout << mean / req_per_min.size() << std::endl;
}
```

Listing 11: `examples/exponential.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).

2.1.2.6 Poisson distribution (docs)

- TODO: The Poisson distribution is closely related to the Exponential distribution, as it generates a number of items given the rate.

2.1.2.7 Geometric distribution (docs)

- TODO: A typical geometric distribution

2.1.2.8 Discrete distribution (docs)

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.

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

int main() {
    std::discrete_distribution<>
        rand_item = {232, 158, 288}; ①

    for (size_t request = 0; request < 1000; request++) {
        switch (rand_item(urng)) {
            case Shirt: ② std::cout << "shirt"; break;
            case Hoodie: ② std::cout << "hoodie"; break;
            case Pants: ② std::cout << "pants"; break;
        }

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

    return 0;
}
```

Listing 12: `examples/TODO.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 [4].

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 (see Section 2.5.3).

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[] 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 ① : matrix)
    transition_matrix.push_back(
        std::discrete_distribution<>(
            weights.begin(), ②  weights.end() ③ )
    );
```

Listing 13: `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 reft(2) and at the end ③ of the vector. This works with dynamic arrays too.

2.2 Data

2.2.1 Manual memory allocation

If you allocate with `new`, you must deallocate with `delete`, you can't mixup them with `malloc()` and `free()`

To avoid manual memory allocation, most of the time it's enough to use the structures in the standard library, like `std::vector<T>`.

2.2.2 Data structures

2.2.2.1 Vectors

You don't have to allocate memory, basically never! You just use the structures that are implemented in the standard library, and most of the time they are enough for our use cases. They are really easy to use.

Vectors can be used as stacks.

2.2.2.2 Deques

Deques are very common, they are like vectors, but can be pushed and popped in both ends, and can b used as queues.

2.2.2.3 Sets

Not needed as much, works like the Python set. Can be either a set (ordered) or an unordered set (uses hashes)

2.2.2.4 Maps

Could be useful. Can be either a map (ordered) or an unordered map (uses hashes)

2.3 I/O

Input output is very simple in C++.

2.3.1 Standard I/O

2.3.2 Files

Working with files is way easier in `C++`

```cpp
#include <fstream>

int main(){
    std::ofstream output("output.txt");
    std::ifstream params("params.txt");

    while (etc ... ) {}
    output.close();
    params.close();
}
```

2.4 Operator overloading

In Listing 6, to generate a random number, `random_device()` ③ and `random_engine()` ⑥ are used like functions, but they aren't functions, they're instances of a `class`. That's because in `C++` you can define how a certain operator (like `+`, `+=`, `<<`, `>>`, `[]`, `()` etc..) should behave when used on a instance of the `class`. It's called **operator overloading**, a relatively common feature:

- in `Python` operation overloading is done by implementing methods with special names, like `__add__()` [5]
- in `Rust` it's done by implementing the `Trait` associated with the operation, like `std::ops::Add` [6].
- `Java` and `C` don't have operator overloading

For example, `std::cout` is an instance of the `std::basic_ostream class`, which overloads the method "`operator<<()`" [7]. The same applies to to file streams.

2.5 Code structure

2.5.1 Classes

- TODO:
  - Maybe constructor
  - Maybe operators? (more like nah)
  - virtual stuff (interfaces)

2.5.2 Structs

- basically like classes, but with everything public by default

2.5.3 Enums

- enum vs enum class
- an example maybe
- they are useful enough to model a finite domain

2.5.4 Inheritance

3 Debugging with `gdb`

It's super useful! Trust me, if you learn this everything is way easier (printf won't be useful anymore)

First of all, use the `-ggdb3` flags to compile the code. Remember to not use any optimization like `-O3`… using optimizations makes the program harder to debug.

```
DEBUG_FLAGS := -ggdb3 -Wall -Wextra -pedantic
```

Then it's as easy as running `gdb ./main`

- TODO: could be useful to write a script if too many args
- TODO: just bash code to compile and run
- TODO (just the most useful stuff, technically not enough):
  - r
  - c
  - n
  - c 10
  - enter (last instruction)
  - b
    - on lines
    - on symbols
    - on specific files
  - clear
  - display
  - set print pretty on

4 Examples

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**.

4.1 First examples

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

4.1.1 A simple MDP `[1100]`

The first MDP $M = (U, X, Y, p, g)$ is such that

- $U = \{\varepsilon\}$[2]
- $X = [0, 1] \times [0, 1]$, each state is a pair ③ of **real** numbers ②
- $Y = [0, 1] \times [0, 1]$
- $p : X \times X \times U \to X = (\mathcal{U}(0, 1), \mathcal{U}(0, 1))$ the transition probability ①
- $g : X \to Y : (r_0, r_1) \mapsto (r_0, r_1)$ outputs the current state ④
- $x_0 = (0, 0)$ is the initial state ③

```cpp
#include <fstream>
#include <random>
#include "../../mocc/mocc.hpp"
const size_t HORIZON = 10;

int main() {
    std::random_device random_device;
    urng_t urng(random_device());
    std::uniform_real_distribution<real_t> uniform(0, 1); ①
    std::vector<real_t ②> state(2, 0); ③
    std::ofstream file("logs");

    for (size_t time = 0; time <= HORIZON; time++) {
        for (auto &r_i : state)
            r_i = uniform(urng); ①

        file << time << ' ';
        for (auto r_i : state)
            file << r_i << ' '; ④
        file << std::endl;
    }

    file.close();
    return 0;
}
```

[2]See Section 1.2.2.1

Listing 14: `software/1100/main.cpp`

4.1.2 MDPs network pt.1 `[1200]`

This example has 2 MDPs M_0, M_1 s.t.

- $M_0 = (U^0, X^0, Y^0, p^0, g^0)$
- $M_1 = (U^1, X^1, Y^1, p^1, g^1)$

M_0 and M_1 are similar to the MDP in Section 4.1.1, with the difference that $U^i = [0, 1] \times [0, 1]$, having $U^i = X^i$, meaning p must be redefined:

$$p^i(x'|x, u) = \begin{cases} 1 \text{ if } x' = u \\ 0 \text{ otherwise} \end{cases} \tag{10}$$

Then the 2 MDPs can be connected

$$\begin{aligned} U^0_{t+1} &= (r_0 \cdot \mathcal{U}(0,1), r_1 \cdot \mathcal{U}(0,1)) \text{ where } g^1(X^1_t) = (r_0, r_1) \\ U^1_{t+1} &= (r_0 + \mathcal{U}(0,1), r_1 + \mathcal{U}(0,1)) \text{ where } g^0(X^0_t) = (r_0, r_1) \end{aligned} \tag{11}$$

Given that $g^i(X^i_t) = X^i_t$ and $U^i_t = X^i_t$, the connection in Equation 11 can be simplified:

$$\begin{aligned} X^0_{t+1} &= (r_0 \cdot \mathcal{U}(0,1), r_1 \cdot \mathcal{U}(0,1)) \text{ where } X^1_t = (r_0, r_1) \\ X^1_{t+1} &= (r_0 + \mathcal{U}(0,1), r_1 + \mathcal{U}(0,1)) \text{ where } X^0_t = (r_0, r_1) \end{aligned} \tag{12}$$

With Equation 12 the code is easier to write, but this approach works for small examples like this one. For more complex systems it's better to design a module for each component and handle the connections more explicitly.

```cpp
/* ... */
const size_t HORIZON = 100;
struct MDP { real_t state[2]; };

int main() {
    /* ... */
    std::vector<MDP> mdps(2, {0, 0});

    for (size_t time = 0; time ≤ HORIZON; time++)
        for (size_t r = 0; r < 2; r++) {
            mdps[0].state[r] =
                mdps[1].state[r] * uniform(urng);
            mdps[1].state[r] =
                mdps[0].state[r] + uniform(urng);
        }

    /* ... */
}
```

Listing 15: `software/1200/main.cpp`

4.1.3 MDPs network pt.2 `[1300]`

This example is similar to the one in Section 4.1.2, with a few notable differences:

- $U^i = X^i = Y^i = \mathbb{R} \times \mathbb{R}$
- the initial states are $x_0^0 = (1, 1), x_0^1 = (2, 2)$
- the connections are slightly more complex.
- no probability is involved

Having

$$p((x_0{}', x_1{}')|(x_0, x_1), (u_0, u_1)) = \begin{cases} 1 \text{ if } \dots \\ 0 \text{ otherwise} \end{cases} \tag{13}$$

The implementation would be

```cpp
/* ... */

int main() {
    /* ... */
    std::vector<MDP> mdps({{1, 1}, {2, 2}});

    for (size_t time = 0; time ≤ HORIZON; time++) {
        mdps[0].state[0] =
            .7 * mdps[0].state[0] + .7 * mdps[0].state[1];
        mdps[0].state[1] =
            -.7 * mdps[0].state[0] + .7 * mdps[0].state[1];

        mdps[1].state[0] =
            mdps[1].state[0] + mdps[1].state[1];
        mdps[1].state[1] =
            -mdps[1].state[0] + mdps[1].state[1];

        /* ... */
    }

    /* ... */
}
```

Listing 16: `software/1300/main.cpp`

4.1.4 MDPs network pt.3 `[1400]`

The original model behaves exactly lik Listing 16, with a different implementation. As an exercise, the reader is encouraged to come up with a different implementation for Listing 16.

4.2 Traffic light `[2000]`

This example models a **traffic light**. The three original versions (`2100`, `2200` and `2300`) have the same behaviour, with a different implementation. The one reported here has the same behaviour of that one, yet it's simpler. Let T be the MDP that describes the traffic light, s.t.

- $U = \{\varepsilon, \sigma\}$ where
  - ε means "do nothing"
  - σ means "switch light"
- $X = \{\text{green}, \text{yellow}, \text{red}\}$
- $Y = X$
- $g(x) = x$
- $p(x'|x, \varepsilon) = \begin{cases} 1 \text{ if } x' = x \\ 0 \text{ otherwise} \end{cases}$
- $p(x'|x, \sigma) = \begin{cases} 1 \text{ if } (x = \text{green} \wedge x' = \text{yellow}) \vee (x = \text{yellow} \wedge x' = \text{red}) \vee (x = \text{red} \wedge x' = \text{green}) \\ 0 \text{ otherwise} \end{cases}$

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, iif the transition is valid (green $\rightarrow$ yellow, yellow $\rightarrow$ red, red $\rightarrow$ green)

```cpp
#include <fstream>
#include <random>
#include "../../mocc/mocc.hpp"
const size_t HORIZON = 1000;
enum Light { GREEN = 0, YELLOW = 1, RED = 2 }; ①

int main() {
    std::random_device random_device;
    urng_t urng(rand_device());
    std::uniform_int_distribution<>rand_interval(60, 120); ②
    Light traffic_light = Light::RED;
    size_t next_switch = rand_interval(urng);
    std::ofstream file("logs");

    for (size_t time = 0; time <= HORIZON; time++) {
        file << time << ' ' << next_switch - time << ' '
             << traffic_light ③ << std::endl;
        if (time < next_switch) continue;
        traffic_light = ④
            (traffic_light == RED
                 ? GREEN
                 : (traffic_light == GREEN ? YELLOW : RED));

        next_switch = time + rand_interval(urng);
    }

    file.close(); return 0;
}
```

Listing 17: `software/2000/main.cpp`

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

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

4.3 Control center

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.

4.3.1 No network `[3100]`

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 ①

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

enum Light { GREEN = 0, YELLOW = 1, RED = 2 }; ②
const size_t HORIZON = 1000; ③
static std::random_device random_device; ④
static urng_t urng = urng_t(random_device()); ④
```

Listing 18: `software/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)… but 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)

```
#pragma once

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

class TrafficLight : public Timed ① {
    std::uniform_int_distribution<> random_interval; ②
    Light l = Light::RED; ③

  public:
    TrafficLight(System *system)
        : random_interval(60, 120),
          Timed(system, 90, TimerMode::Once) {} ④

    void update(U) override { ⑤
        l = (l == RED ? GREEN : (l == GREEN ? YELLOW : RED));
        timer.set_duration(random_interval(urng)); ⑦
    }

    Light light() { return l; } ⑧
};
```

Listing 19: `software/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.

```
#include <fstream>

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

int main() {
    std::ofstream file("logs");
    System system; ①
    Stopwatch stopwatch; ②
    TrafficLight traffic_light(&system); ③
    system.addObserver(&stopwatch);

    while (stopwatch.elapsed() <= HORIZON) {
        file << stopwatch.elapsed() << ' '
             << traffic_light.light() << std::endl;
        system.next(); ④
    }

    file.close();
    return 0;
}
```

Listing 20: `software/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`.

4.3.2 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.

4.3.2.1 No faults `[3200]`

```
/* ... */

class ControlCenter
    : public Timed, public Notifier<Payload> ① {
    std::uniform_int_distribution<> random_interval;
    Light l = Light::RED;

  public:
    ControlCenter(System *system)
        : random_interval(60, 120),
          Timed(system, 90, TimerMode::Once) {}

    void update(U) override {
        l = (l == RED ? GREEN : (l == GREEN ? YELLOW : RED));
        notify(l); ①
```

```cpp
        timer.set_duration(random_interval(urng));
    }

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

Listing 21: `software/3200/control_center.hpp`

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

```cpp
/* ... */

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

  public:
    void update(Message message) override {
        ② notify(l = message ①);
    }

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

Listing 22: `software/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`).

```cpp
/* ... */
#include "../../mocc/alias.hpp"

/* ... */
STRONG_ALIAS(Payload, Light);
STRONG_ALIAS(Message, Light);
```

Listing 23: `software/3200/parameters.hpp`

The `STRONG_ALIAS`es 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 they are different is different from the type they alias (`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.

```
/* ... */

class Network : public Timed,
                public Buffer<Payload>, ①
                public Notifier<Message> {

  public:
    Network(System *system)
        : Timed(system, 0, TimerMode::Once) {}

    void update(Payload payload) override {
        if (buffer.empty())
            timer.set_duration(2);
        Buffer<Payload>::update(payload);
    }

    void update(U) override {
        if (!buffer.empty()) {
            notify((Light)buffer.front());
            buffer.pop_front();
            if (!buffer.empty())
                timer.set_duration(2);
        }
    }
};
```

Listing 24: `software/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...

```
/* ... */

class Monitor : public Recorder<Payload>,
                public Recorder<Light>,
                public Observer<> {

    int time = 0; bool messages_lost = false;

  public:
    Monitor() : Recorder<Payload>(RED), Recorder<Light>(RED)
{}

    void update() override {
        if (Recorder<Payload>::record !=
            Recorder<Light>::record)
            time++;
        else
            time = 0;

        if (time > 3)
            messages_lost = true;
    }

    bool is_valid() { return !messages_lost; }
};
```

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

The `Monitor` is a compoment 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).

4.3.2.2 Faults & no repair `[3300]`

```
/* ... */

class Network : public Timed,
                public Buffer<Payload>,
                public Notifier<Message> {
    std::bernoulli_distribution random_fault; ①

  public:
    /* ... */

    void update(U) override {
        if (!buffer.empty()) {
            if (!random_fault(urng)) ②
                notify((Light)buffer.front());
            buffer.pop_front();
            if (!buffer.empty())
                timer.set_duration(2);
        }
    }
};
```

Listing 26: `software/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.

4.3.3 Faults & repair [3400]

```
/* ... */

class Network : public Timed,
                public Buffer<Payload>,
                public Notifier<Message> {
    std::bernoulli_distribution random_fault;
    std::bernoulli_distribution random_repair; ①

  public:
    /* ... */

    void update(U) override {
        if (!buffer.empty()) {
            if (!random_fault(urng)||random_repair(urng)) ②
                notify((Light)buffer.front());
            buffer.pop_front();
            if (!buffer.empty())
                timer.set_duration(2);
        }
    }
};
```

Listing 27: `software/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.

4.3.4 Faults & repair + correct protocol [3500]

```
/* ... */

class Network : public Timed,
                public Buffer<Payload>,
                public Notifier<Message>,
                public Notifier<Fault> ① {
  public:
    /* ... */

    void update(U) override {
        if (!buffer.empty()) {
            if (random_fault(urng)) {
                if (random_repair(urng))
                    Notifier<Message>::notify(
                        (Light)buffer.front());
                else
                    Notifier<Fault>::notify(true); ②
            } else {
                Notifier<Message>::notify(
                    (Light)buffer.front());
            }

            buffer.pop_front();
            if (!buffer.empty())
                timer.set_duration(2);
        }
    }
};
```

Listing 28: `software/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.

```
/* ... */

class TrafficLight : public Observer<Fault>, ①
                     public Observer<Message>,
                     public Notifier<Light> {
    /* ... */

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

Listing 29: `software/3500/traffic_light.hpp`

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

```
System system;
Monitor monitor;
Network network(&system);
Stopwatch stopwatch;
TrafficLight traffic_light;
ControlCenter control_center(&system);

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

Listing 30: connections in `software/3500/main.cpp`

4.4 Statistics

4.4.1 Expected value [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.

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

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

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

    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},
        };

    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 0;
}
```

Listing 31: software/4100/main.cpp

4.4.2 Probability [4200]

This example behaves like the previous one, but uses the Monte Carlo method [2] to calculate the probability the cost is less than a certain value

```cpp
/* ... */

const size_t ITERATIONS = 10000;

int main() {
    Stat cost_stat;
    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;
        }

        cost_stat.save(costs);
        if (costs < 100)
            less_than_100_count++;
    }

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

    file.close();
    return 0;
}
```

Listing 32: software/4200/main.cpp

4.5 Development process simulation

An MDP can be implemented by using a **transition matrix** (like in Section 1.2.2.1). The simplest implemenation can be done by using a `std::discrete_distribution` by using the trick in Listing 8.

4.5.1 Random transition matrix `[5100]`

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

```cpp
const size_t HORIZON = 20, STATES_SIZE = 10;

int main() {
    std::random_device random_device;
    urng_t urng(random_device());
    auto random_state = ①
        std::uniform_int_distribution<>(0, STATES_SIZE - 1);
    std::uniform_real_distribution<> random_real_0_1(0, 1);

    std::vector<std::discrete_distribution<>>
        transition_matrix(STATES_SIZE); ②
    std::ofstream log("log.csv");

    for (size_t state = 0; state < STATES_SIZE; state++) {
        std::vector<real_t> weights(STATES_SIZE); ③
        for (auto &weight : weights)
            weight = random_real_0_1(urng);

        transition_matrix[state] = ④
            std::discrete_distribution<>(weights.begin(),
                                         weights.end());
    }

    size_t state = random_state(urng);
    for (size_t time = 0; time <= HORIZON; time++) {
        log << time << " " << state << std::endl;
        state = transition_matrix[state ⑤ ](urng); ⑥
    }

    log.close();
    return 0;
}
```

Listing 33: `software/5100/main.cpp`

A **transition matrix** is a `vector<discrete_distribution<>>` ② just like in Listing 8. 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.

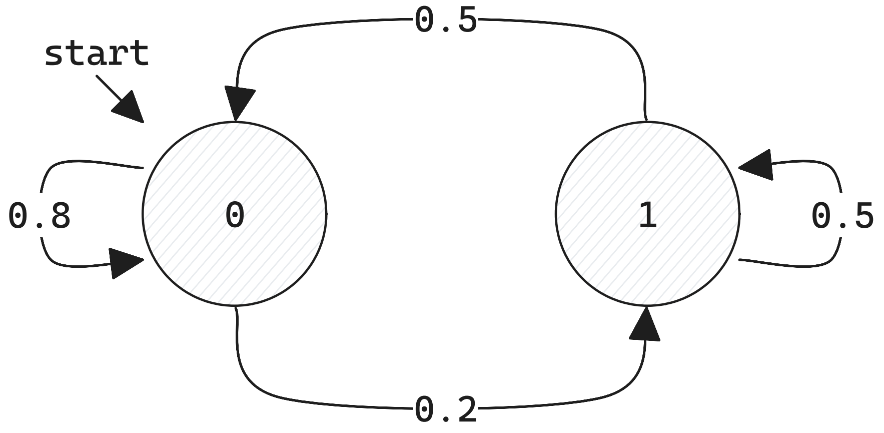

Figure 5: A simple Markov Chain

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 8, 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.

4.5.2 `[5200]` Software development & error detection

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 <...>

using real_t = double;
const size_t HORIZON = 800, PHASES_SIZE = 3;

enum Outcome ① {
    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)
            phases_error_distribution.push_back(
                ② 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);
    }

    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]++;

        if (progress[phase] == 4) {
            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;
        }
    }

    return 0;
}
```

Listing 34: `software/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.

4.5.3 Optimizing costs for the development team `[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).

4.5.4 Key performance index `[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).

4.6 Complex systems

4.6.1 Insulin pump `[6100]`

4.6.2 Buffer `[6200]`

4.6.3 Server `[6300]`

5 MOCC library

Model CheCking library for the exam

5.1 Design

Basically: the "Observer Pattern" [8] can be used to implement MDPs, because a MDP is like an entity that "is notified" when something happens (receives an input, in fact, in the case of MDPs, another name for input is "action"), and notifies other entities (gives an output, or reward).

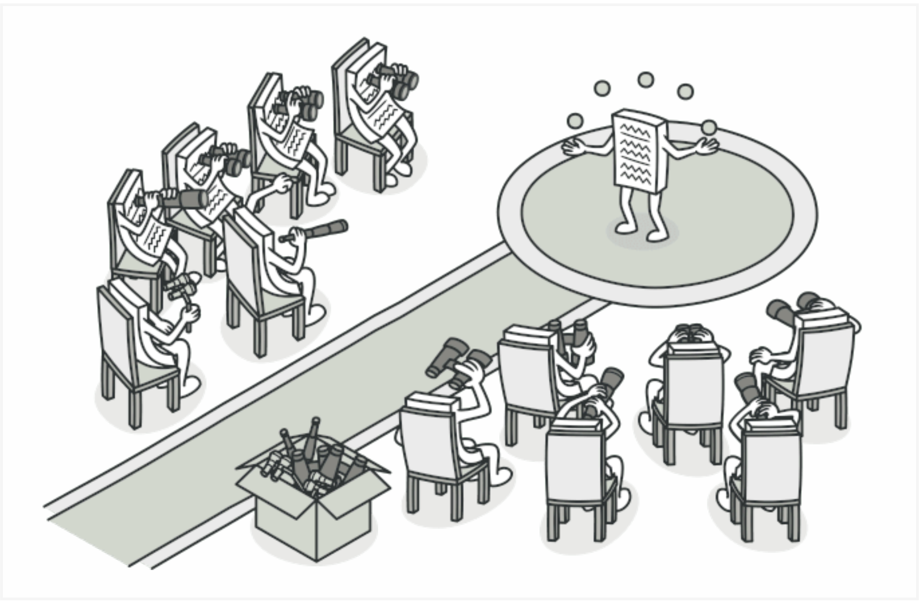

Figure 6: `https://refactoring.guru/design-patterns/observer`

By using the generics (templates) in `C++` it's possible to model type-safe MDPs, whose connections are easier to handle (if an entity receives inputs of type `Request`, it cannot be mistakenly connected to an entity that gives an output of type `Time`).

5.2 `mocc`

```
using real_t = double;
```

The `real_t` type is used as an alias for floating point numbers to ensure the same format is used everywhere in the library.

```
using urng_t = std::default_random_engine;
```

The `urng_t` type is used as an alias for `std::default_random_engine` to make the code easier to write.

5.3 `math`

```
class Stat
```

The `Stat` class is used to calculate the mean and the standard deviation of a set of values (as discussed in Section 1.3.1 and Section 1.3.2)

```
void save(real_t x);
```

The `save()` method is used to add a value to the set of values. The mean and the standard deviation are automatically updated when a new value is saved.

```
real_t mean() const;
```

Returns the precalculated mean.

```
real_t stddev() const;
```

Returns the precalculated standard deviation.

Example

```
Stat cost_stat;

cost_stat.save(302);
cost_stat.save(305);
cost_stat.save(295);
cost_stat.save(298);

std::cout
  << cost_stat.mean() << " "
  << cost_stat.stddev() << std::endl;
```

5.4 time

```
STRONG_ALIAS(T, real_t)
```

The `T` is the type for the **time**, it's reperesented as a `real_t` to allow working in smaller units of time (for exapmle, when the main unit of time of the simulation is the *minute*, it could still be useful to work with *seconds*). `T` is a **strong alias**, meaning that if a MDP takes in input `T`, it cannot be connected to a MDP that gives in output a simple `real_t`.

```
class Stopwatch : public Observer<>, public Notifier<T>
```

A `Stopwatch` starts at time `0`, and each iteration of the system it increments it's time counter by Δ. It can be used to measure time from a certain point of the simulation (it can be at any point of the simulation). It sends a notification with the elapsed time at each iteration.

```
Stopwatch(real_t delta = 1);
```

The default Δ for the `Stopwatch` is `1`, but it can be changed. Usually, a `Stopwatch` is connected to a `System`.

```
real_t elapsed();
```

Returns the time elapsed since the `Stopwatch` was started.

```
void update() override;
```

This method **must** be called to update the `Stopwatch`. It is automatically called when the `Stopwatch` is connected to a `System`, or, more generally, to a `Notifier<>`.

Example

```
System system;
Stopwatch s1, s2(2.5);

size_t iteration = 0;
system.addObserver(&s1);

while (s1.elapse() < 10000) {
    if (iteration == 1000) system.addObserver(&s2);
    system.next(); iteration++;
}
```

```
std::cout << s1.elapsed() <<' '<< s2.elapsed() <<
std::endl;
```

```
enum class TimerMode { Once, Repeating }
```

A `Timer` can be either in `Repeating` mode or in `Once` mode:

- In `Repeating` mode, everytime the timer hits 0, it resets
- In `Once` mode, when the timer hits 0, it stops

```
class Timer : public Observer<>, public Notifier<>
```

A `Timer` starts with a certain duration. At every iteration the duration decreases by Δ. When a `Timer` hits 0, it sends a notification to its subscribers (with no input value).

```
Timer(real_t duration, TimerMode mode, real_t delta = 1);
```

A `Timer` requires the starting duration and it's mode. It's more useful to use the `Once` mode if the duration is different at each reset, this way it can be set manually.

```
void set_duration(real_t time);
```

Sets the current duration of the `Timer`. It's useful when the duration is generated randomly each time the `Timer` hits 0.

```
void update() override;
```

This method must be called to updated the time of the `Timer`. Generally the `Timer` is connected to a `System`.

Example

```
TODO: example
```

5.5 `alias`

```
template <typename T> class Alias
```

The `class Alias` is used to create **strong aliases** (a strong alias is a type that can be used in place of its underlying type, except in templates, as its considere a totally different type).

> ```
> Alias() {}
> ```
>
> It initialized the value for the underlying type to it's default one.
>
> ```
> Alias(T value)
> ```
>
> It initialized the underlying type with a certain value. Useful when the underlying type needs complex initialization. It also allows to assign a value of the underlying type (e.g. `Alias<int> a_int = 5;`)
>
> ```
> operator T() const
> ```
>
> Allows the `Alias<T>` to be casted to `T` (e.g. `Alias<int> a_int = 5; int v = (int)a_int;`). The casting doesn't need to be explicit.

```
STRONG_ALIAS(ALIAS, TYPE)
```

The `STRONG_ALIAS` macro is used to quickly create a strong alias. The `Alias<T>` class is never used directly.

5.6 `observer`

```
template <typename ... T> class Observer
```

- TODO

5.7 `notifier`

```
template <typename ... T> class Notifier
```

- TODO

5.8 Auxiliary

```
template <typename T> class Recorder : public Observer<T>
```

```
class Client : public Observer<U ... >,
               public Notifier<Observer<U ... > *, T>
```

- TODO (+ `using Host`)

```
class Server : public Observer<Observer<U ... > *, T>
```

- TODO (+ `using Host`)

```
class System : public Notifier<>
```

- TODO

6 Practice

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) ①
    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 35: `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" ①
#include "../../mocc/mocc.hpp" ②

STRONG_ALIAS(ProjInit, real_t) ③
STRONG_ALIAS(TaskDone, real_t) ③
STRONG_ALIAS(EmplCost, real_t) ③

static ④ real_t A, B, C, D, F, G;
static size_t N, W, HORIZON = 100000;

#endif
```

Listing 36: `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; ①
Stopwatch stopwatch; ②

system.addObserver(&stopwatch); ③

while (stopwatch.elapsed() ≤ HORIZON) ④
    system.next(); ⑤
```

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

6.1 Development team (time & cost)

6.1.1 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 37: `practice/1/employee.hpp`

6.1.2 Director

6.2 Task management

6.2.1 Worker

6.2.2 Generator

6.2.3 Dispatcher (not the correct name)

6.2.4 Manager (not the correct name)

6.3 Backend load balancing

6.3.1 Env

6.3.2 Dispatcher, Server and Database

6.3.3 Response time

6.4 Heater simulation

7 Extras

7.1 VDM (Vienna Development Method)

"The Vienna Development Method (VDM) is one of the longest established model-oriented formal methods for the development of computer-based systems and software. It consists of a group of mathematically well-founded languages and tools for expressing and analyzing system models during early design stages, before expensive implementation commitments are made. The construction and analysis of the model help to identify areas of incompleteness or ambiguity in informal system specifications, and provide some level of confidence that a valid implementation will have key properties, especially those of safety or security. VDM has a strong record of industrial application, in many cases by practitioners who are not specialists in the underlying formalism or logic. Experience with the method suggests that the effort expended on formal modeling and analysis can be recovered in reduced rework costs arising from design errors." [9]

7.1.1 It's cool, I promise

- Alloy? Maybe it's a good alternative, haven't tried it enough
- VDM is basically OCL (Object Constraint Language) but better.

7.1.2 VDM++ to design valid UMLs

7.2 Advanced testing techinques (in `Rust` & `C`)

- TODO: cite "Rust for Rustaceans"
- TODO: unit tests aren't the only type of test

7.2.1 Mocking (mockall)

7.2.2 Fuzzying (cargo-fuzz)

7.2.3 Property-based testing

7.2.4 Test augmentation (Miri, Loom, Valgrind)

7.2.5 Performance testing

- Rust is very focused on performance
- TODO: non-functional requirements

7.3 Model checking with Bevy (`Rust`)

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

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

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

[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] “Observer Pattern.” [Online]. Available: https://refactoring.guru/design-patterns/observer

[9] “The Vienna Development Method.” [Online]. Available: https://www.overturetool.org/method/