Imię i nazwisko: | | | | | | | | | | | | | | | | | | | | | | | | | | **24Z kol 2**

Proszę wpisywać odpowiedzi w miejscach na to przeznaczonych

Zadanie 1 (2pkt)

Proszę odpowiedzieć na poniższe pytania:

- Czy (i ewentualnie jak) można mierzyć jakość testu?

- Dlaczego przyjmuje się, że „idealny" test powinien być deterministyczny? Podaj przykład problemu i jego rozwiązania.

Zadanie 2 (2pkt)

Dana jest funkcja:

```
std::optional<Value> Backend::handler(const Request* r, std::string_view name) {
  if ((r == nullptr) || name.empty())
    throw InvalidArgument();
  const auto value = db.contains(name) ? db.load(name) : buildDefaultFor(name);
  if (value.generation < r->generation())
    return std::nullopt;
  return value;
}
```

Ile przypadków testowych należy napisać, by uzyskać:

- 100% pokrycia linii kodu,
- 100% pokrycia decyzji w kodzie.

Podaj elementy klas równoważności argumentów konieczne do uzyskania wspomnianych pokryć.

Zadanie 3 (2pkt)

Co jest nie tak z poniższym kodem?

```
int Client::upload(const Item& i, bool apply, Processing p)
{
  // sprawdzamy polaczenie
  if (isConnected() && isAuthorized() && (itemsQueue.size() == 0) && ((security == nullptr)
      || security->allowsUploads() || security->isAdmin()))
  {
    auto tmp = i.elements.cache[i.id];
    switch (p) {
        case Processing::Par:
            if (apply)
                cache.insert(i);
            const auto response = buildResponse(i, tmp * 2);
            if (tmp < i.limit)
                return -3;
            return asyncResponse(response);
        case Processing::Seq:
            if (apply) {
                cache.insert(i);
            }
            if (tmp < i.limit) {
                return -3;
            }
            return sendResponse(buildResponse(i, itemsQueue.size() + tmp));
        default: return -2;
    }
    logger.log("Response␣sent");
  }
  else
  {
    return -4;
  }
}
```

Proszę opisać **jak** należałoby poprawić ten kod (jakie operacje i w jaki sposób wykonane, kod nie jest konieczny).

Uwagi do prowadzącego (K. Grochowskiego):

Imię i nazwisko: | | | | | | | | | | | | | | | | | | | | | | | | | | **24Z kol 2**

Proszę wpisywać odpowiedzi w miejscach na to przeznaczonych

Zadanie 4 - współbieżne wzorce projektowe (2pkt)

Czy w programie pokazanym niżej występuje wyścig? [ ]

```
using namespace std;
int main () {
    Queue q;
    Reader r1(q), r2(q); Writer w1(q), w2(q);
    thread t1(ref(r1)); thread t2(ref(r2)); thread t3(ref(w1)); thread t4(ref(w2));
    std::this_thread::sleep_for(chrono::milliseconds(2000));
    r1.finish(); r2.finish(); w1.finish(); w2.finish();
    t1.join(); t2.join(); t3.join(); t4.join();
    return 0;
}
```

Popraw wydajność na platformach współbieżnych. Obiekty typu `Data` są duże. W kodzie występują obiekty `std::ref(x)`, są to obiekty pomocnicze, które zapobiegają tworzeniu kopii obiektu, więc np. `thread t(ref(p));` oznacza, że konstruktor `std::thread` nie wykona kopii obiektu `p`. Nie jest to kluczowe do znalezienia rozwiązania. Posługujemy się także listą dwukierunkową `std::list`, z operacjami `push_back` –wstawiania na koniec, `pop_front` – usuwanie początkowego elementu, `front` – zwraca początkowy element listy.

```
using Data = int; //Data to paczka danych do przetwarzania
const Data EMPTY = -1; //reprezentuje brak danych
struct Queue {
    std::list<Data> dane_;
    std::mutex mutex_;
    bool isEmpty() {
        return dane_.empty();
    }
    Data read() {
        Data out = EMPTY;
        if(!dane_.empty() ) {
            out = dane_.front();
            dane_.pop_front();
        }
        return out;
    }
    void write(Data d) {
        dane_.push_back(d);
    }
};
class Reader {
public:
    Reader(Queue& q) : finish_(false), queue_(q) {}
    void finish() { finish_ = true; }
    void operator()() {
      while(!finish_) {
        std::lock_guard<std::mutex> lock(queue_.mutex_);
        if(! queue_.isEmpty() ) {
          Data d = queue_.read();
          //ten komentarz oznacza algorytm, który wykonuje się długo i lokalnie, przetwarza 'd'
        }
      }
    }
private:
    Queue& queue_;
    volatile bool finish_;
};
class Writer {
public:
    Writer(Queue& q) : finish_(false), queue_(q) {}
    void finish() { finish_ = true; }
    void operator()() {
      while(!finish_) {
        std::lock_guard<std::mutex> lock(queue_.mutex_);
        //ten komentarz oznacza algorytm, który wykonuje się lokalnie i długo, na koniec inicjuje obiekt 'd'
        Data d;
        queue_.write(d);
      }
    }
private:
    Queue& queue_;
    volatile bool finish_;
};
```

Zadanie 5 - współbieżne wzorce projektowe (2pkt)

Zliczamy obiekty `Data`, które mają odpowiednie właściwości w 2 wątkach. Popraw wydajność funkcji `countProper`. Nie można zmieniać funkcji `isProper`, która liczy się długo.

```
using Data; //zakladamy, ze Data jest to duzy obiekt
bool isProper(const Data& d) {
  //bardzo dlugi algorytm lokalny
}
```

```
struct InData {
    InData(const vector<Data>& r) : records_(r), counter_(0) {}
    const vector<Data>& records_;
    mutex m_;
    int counter_;
};
struct Thread {
    Thread(InData& d) : data_(d), currentIdx_(0) {}

    void setIdx(int i) { currentIdx_ = i; }
    void operator()() {
        std::lock_guard<std::mutex> lock(data_.m_);
        if(isProper(data_.records_[currentIdx_])) {
            data_.counter_ += 1;
        }
    }
private:
    int currentIdx_;
    InData& data_;
};

int countProper(const vector<Data>& records) {
    InData input(records);
    Thread t1(input);
    Thread t2(input);

    int i = 0; //indeks
    while( i < records.size() ) {
        t1.setIdx(i);
        ++i;
        thread thrd1( ref(t1) );
        if( ++i < records.size() ) {
            t2.setIdx(i);
            ++i;
            thread thrd2( ref(t2) );
            thrd2.join();
        }
        thrd1.join();
    }
    return input.counter_;
  }
```

Uwagi do prowadzącego (R. Nowaka):