Automatisation de tests asynchrones

Gabriel Aubut-Lussier

Gabriel Aubut-Lussier

Druide

Plan

- Principes généraux appliqués aux tests automatisés
- Asynchronicité
- Survol des problèmes asynchrones typiques
- Techniques pour tester un système asynchrone
- Comparaison des techniques
- Conclusion

Retour sur une présentation antérieure

- Conclusions de la présentation "To catch a mocking bug!" par Jean-Michel Carter :
  - Legacy code is Code without test
  - Legacy code is a nightmare to refactor
  - Legacy code is liable to be broken in the future (Beyoncé rule)
  - Tests gives you confidence to refactor
  - TDD (Test Driven Development) helps to achieve good coverage and prevents the creation of brand new legacy code
  - Catch(2) gives you an easy framework to start doing tests

Principes supplémentaires

- Tester seulement la partie publique d'un API
- Prouver le bon fonctionnement du test (il doit échouer au moins une fois)
- Minimiser le temps d'exécution des tests
- Supprimer les tests qui freinent l'évolution sans jamais détecter de régressions
- Valider les binaires avant qu'ils soient déployés (élimine des étapes manuelles dans le processus d'assurance qualité)

Asynchronicité

```cpp
using Callback = std::function<void(int)>;

int f();
void f(Callback);
std::future<int> g();

int main()
{
  // Exemple d'invocation de fonction synchrone
  [[maybe_unused]] auto resultat = f();

  // Exemple d'invocation asynchrone avec une fonction de rappel
  f([](auto resultat) {/* […] */});

  // Exemple d'invocation asynchrone avec une promesse
  auto promesse = g();
  [[maybe_unused]] auto resultat2 = promesse.get();
}
```

Modèle actions/événements

- Le système reçoit des actions
- Le système produit des événements
- Les actions doivent être traitées séquentiellement
- Les événements doivent être consommés séquentiellement
- Un système synchrone peut être transformé en système asynchrone grâce à ce modèle

Modèle actions/événements

```
// Actions
struct MouseClick {};
struct MouseDown {};
struct MouseMove {};
struct MouseUp {};
```

```
// Events
struct OnMouseClick {
  unsigned int clickCount;
};
struct OnMouseDown {};
struct OnMouseMove {
  bool pressInitiated;
  bool pressed;
};
struct OnMouseUp {};
```

Problèmes asynchrones typiques

- Les assertions ne peuvent pas être exécutées dans le scope du test

Problèmes asynchrones typiques

- Les assertions ne peuvent pas être exécutées dans le scope du test

```
// Syntaxe fictive inspirée de frameworks JavaScript
SCENARIO("Scenario 1", std::function<void()> done)
{
  traitementAsynchrone{[done]() {
    // Ce scope est potentiellement
    // exécuté après la fin du scope parent
    done();
  }};
}
```

Problèmes asynchrones typiques

- Les assertions ne peuvent pas être exécutées dans le scope du test

```
// Syntaxe fictive inspirée de frameworks JavaScript
SCENARIO("Scenario 1")
{
  return std::async(std::launch::async, []() {
    REQUIRE(traitementSynchrone());
  });
}
```

Problèmes asynchrones typiques

- Les assertions ne peuvent pas être exécutées dans le scope du test
- Le déroulement du test dépend de résultats intermédiaires asynchrones

Problèmes asynchrones typiques

- Les assertions ne peuvent pas être exécutées dans le scope du test
- Le déroulement du test dépend de résultats intermédiaires asynchrones

```
SCENARIO("Scenario 1")
{
  auto future = LancerTraitement();
  // Attente implicite
  auto resultat = future.get();
  // Assertions
  REQUIRE(resultat.aUnePropriete());
}
```

Problèmes asynchrones typiques

- Les assertions ne peuvent pas être exécutées dans le scope du test
- Le déroulement du test dépend de résultats intermédiaires asynchrones
- Une action déclenche plusieurs effets observables dont l'ordre n'est pas déterministe

Problèmes asynchrones typiques

- Les assertions ne peuvent pas être exécutées dans le scope du test
- Le déroulement du test dépend de résultats intermédiaires asynchrones
- Une action déclenche plusieurs effets observables dont l'ordre n'est pas déterministe
- L'assertion consiste à prouver qu'un événement n'aura jamais lieu dans le futur

Quelques techniques pour tester un système asynchrone

- Comparaison de journaux d'événements
- Consommation d'événements
- Simulation de l'aggrégation des événements

Étude de cas

```
class Button
{
public:
  using Action = std::variant<MouseClick, MouseDown, MouseMove, MouseUp>;
  using Event = std::variant<OnMouseClick, OnMouseDown, OnMouseMove, OnMouseUp>;
  using EventHandler = std::function<void(Event)>;

public:
  Button();
  Button(EventHandler);

  void mouseClick();
  void mouseDown();
  void mouseMove();
  void mouseUp();

private:
  // […] Implementation details […]
};
```

Système asynchrone

```
template <typename T>
class AsyncSystem
{
public:
  using Action = typename T::Action; // std::variant<…>
  using Event = typename T::Event; // std::variant<…>
  using EventHandler = std::function<void(Event)>;

  AsyncSystem(EventHandler consumeFct);
  ~AsyncSystem();

  void Apply(Action a);
  void InterruptAndWait();
  void WaitForAllTasks();

  // […] membres privés sur la slide suivante
};
```

Système asynchrone

```
template <typename T>
class AsyncSystem
{
  // […] membres publics sur la slide précédente
private:
  struct Tasks {
    bool closeInitiated{false};
    std::deque<std::function<void()>> taskQueue;
  };
  mutable VariableCondition<Tasks> asyncThreadTasks;
  T t;
  std::thread thread;
};
```

Comparaison d'événements

Comparaison d'événements

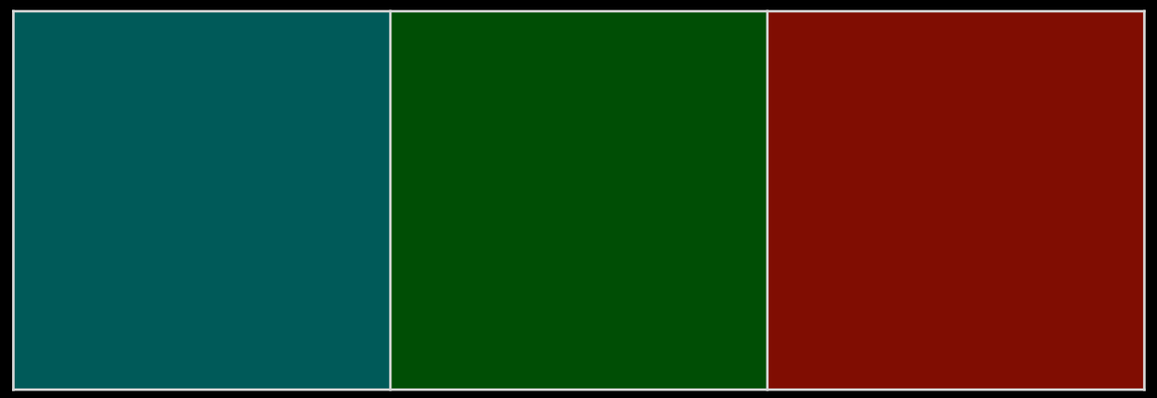
```
std::vector<Button::Event> evenements;
AsyncSystem<Button> b{[&evenements](Button::Event e) {
  evenements.push_back(e);
}};
```

```
WHEN("it is clicked") {
  b.Apply(MouseClick{});
  b.WaitForAllTasks(); // Attente explicite

  THEN("the clickCount is adjusted") {
    REQUIRE(evenements.size() == 3ul);
    REQUIRE(evenements[0].index() == Button::Event{OnMouseDown{}}.index());
    REQUIRE(evenements[1].index() == Button::Event{OnMouseUp{}}.index());
    REQUIRE(evenements[2].index() == Button::Event{OnMouseClick{}}.index());
    REQUIRE(std::get<OnMouseClick>(evenements[2]).clickCount == 1);
  }
}
```

Problèmes avec la comparaison d'événements

- Les tests sont très verbeux
- Les tests sont couplés avec les détails d'implémentation
- Les tests sont difficiles à écrire lorsque l'ordre des événements n'est pas déterministe
- Requiert de l'attente explicite

Consommation d'événements



Événements attendus

Événements consommés ... ... ✔

Utilitaire pour faciliter les tests asynchrones

```
template <typename T>
class AsyncSystemTestRig : public AsyncSystem<T>
{
public:
  using Action = typename T::Action;
  using Event = typename T::Event;

public:
  AsyncSystemTestRig();
  ~AsyncSystemTestRig();

  template <typename E>
  std::optional<E> Get(std::chrono::seconds delai =
std::chrono::seconds{2});

private:
  VariableCondition<std::vector<Event>> eventQueue;
};
```

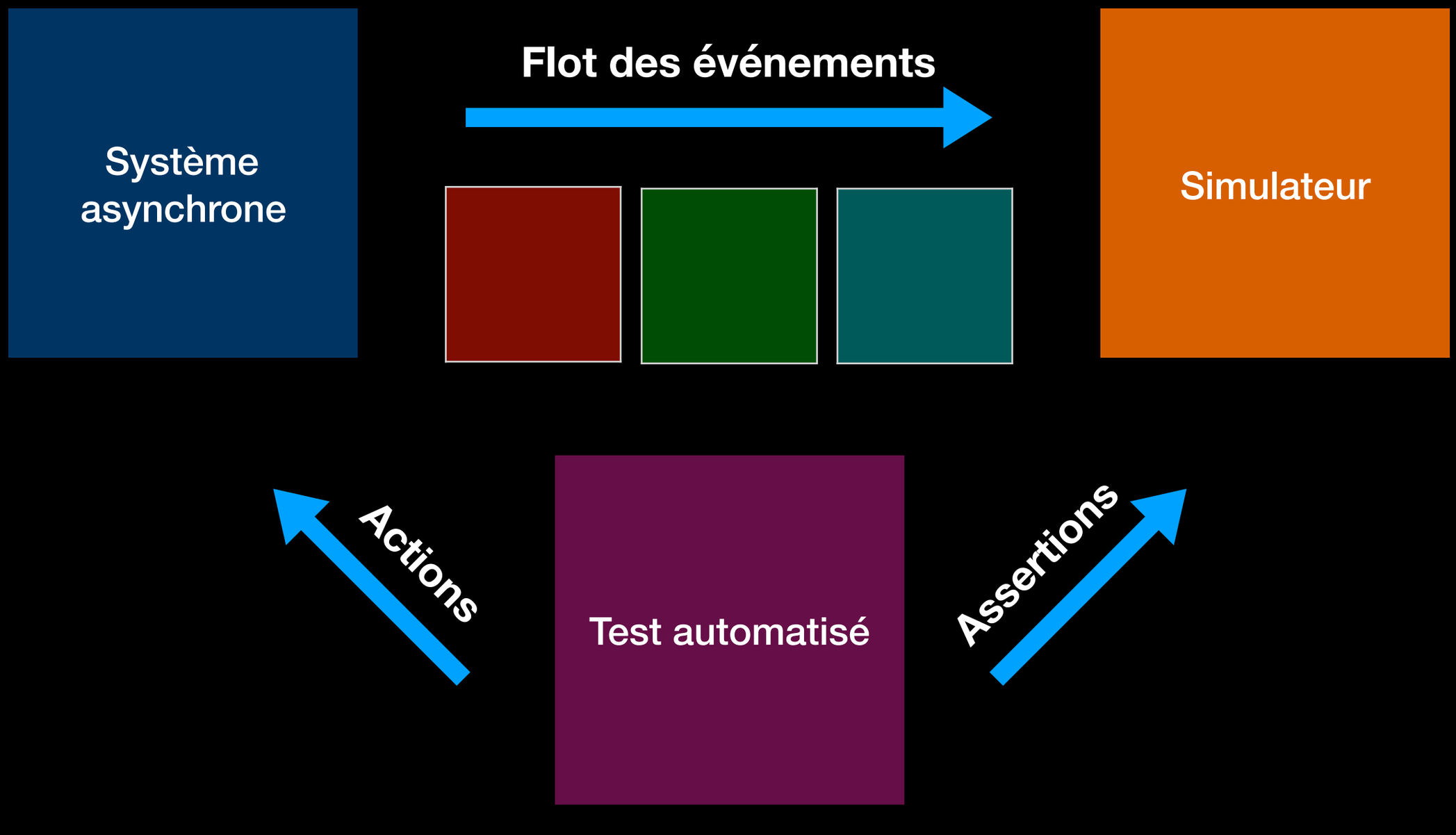
Consommation d'événements

```
AsyncSystemTestRig<Button> b;

WHEN("it is clicked") {
  b.Apply(MouseClick{});

  THEN("the clickCount is adjusted") {
    auto onClickOpt = b.Get<OnMouseClick>(); // Attente implicite
    REQUIRE(onClickOpt);
    REQUIRE(onClickOpt->clickCount == 1);
  }
}
```

Problèmes avec la consommation d'événements

- Les tests sont modérément verbeux
- Les tests sont modérément couplés avec les détails d'implémentation
- Les tests sont moins difficiles à écrire lorsque l'ordre des événements n'est pas déterministe
- Élimine la nécessité d'attendre explicitement

Simulation de l'aggrégation des d'événements



Simulation de l'aggrégation des événements

```
class ButtonControlledMissileSystem
{
public:
  ButtonControlledMissileSystem();

  bool areMissilesArmed() const;
  unsigned int getMissilesFired() const;

  template <typename T> void Apply(T t);
  void Apply(Button::Action a);
  void WaitForAllTasks();

private:
  AsyncSystem<Button> b;
  bool missilesArmed = false;
  unsigned int missilesFired = 0;
};
```

Simulation de l'aggrégation des événements

```
ButtonControlledMissileSystem b;

WHEN("its button is clicked") {
  b.Apply(MouseClick{});
  b.WaitForAllTasks();

  THEN("a missile is fired") {
    REQUIRE(b.getMissilesFired() == 1);
  }
}
```

Problèmes avec la consommation d'événements

- Les tests sont très peu verbeux
- Les tests sont faiblement couplés avec les détails d'implémentation
- Le déterminisme de l'ordre des événements n'a pas d'incidence
- Réintroduit la nécessité d'attendre explicitement

Comparaison des méthodologies

	Tests synchrones	Comparaison de séquences d'événements	Consommation d'événements	Simulation de l'aggrégation des événements
Difficulté de mise en place de l'infrastructure pour réaliser les tests	Non applicable	Faible	Modérée	Liée au degré de fidélité de la simulation
Niveau de déterminisme requis au niveau du séquencement des événements	Non applicable	Élevé	Faible	Quasiment nul
Est-il aisé de valider les champs qui accompagnent un événement?	Oui	Non	Oui	Non
Niveau de verbosité des tests rédigés	Faible	Élevée	Modérée	Faible
Niveau de couplage avec les détails d'implémentation	Modéré	Élevé	Modéré	Faible
Faut-il attendre explicitement la fin du traitement de la séquence d'actions?	Non applicable	Oui	Non	Oui

Bénéfices collatéraux

- L'investissement dans une suite de tests d'intégration a produit des bénéfices qui n'avaient pas été anticipés :
    - Profilage de l'application grandement facilité par l'exécution de la suite de tests automatisés
        - Étude des allocations
        - Étude du partage de la mémoire entre processus
- Les sanitizers gagnent en efficacité lorsque combinés avec les tests
- Découverte des dépendances chargées paresseusement
- Exécution de tests de charge rudimentaires

Conclusion

- La comparaison d'événements ne nécessite que peu d'infrastructure pour être mise en place
- La consommation d'événements et la simulation de l'aggrégation des événements sont des approches supérieures et complémentaires
- Les bénéfices collatéraux sont nombreux et majeurs