ACCU 2018

Mocking Frameworks - considered harmful
Mocking Frameworks considered, harmful?!
mocking Frameworks - considered harmful
Mocking & Frameworks - considered harmful

INSTITUTE FOR SOFTWARE

Prof. Peter Sommerlad
Director of IFS

April 2018

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Cevelop
Your C++ deserves it
Download IDE at:
www.cevelop.com

The Addison-Wesley Signature Series

A Martin Fowler Signature Book

xUnit Test Patterns

Refactoring Test Code

Gerard Meszaros

Foreword by Martin Fowler

Robert C.

Working Effectively with Legacy Code

Michael C. Feathers

Martin Fowler

Contents

Mocks Aren't Stubs

The term 'Mock Objects' has become a popular one to describe special case objects that mimic real objects for testing. Most language environments now have frameworks that make it easy to create mock objects. What's often not realized, however, is that mock objects are but one form of special case test object, one that enables a different style of testing. In this article I'll explain how mock objects work, how they encourage testing based on behavior verification, and how the community around them uses them to develop a different style of testing.

---

02 January 2007

Martin Fowler

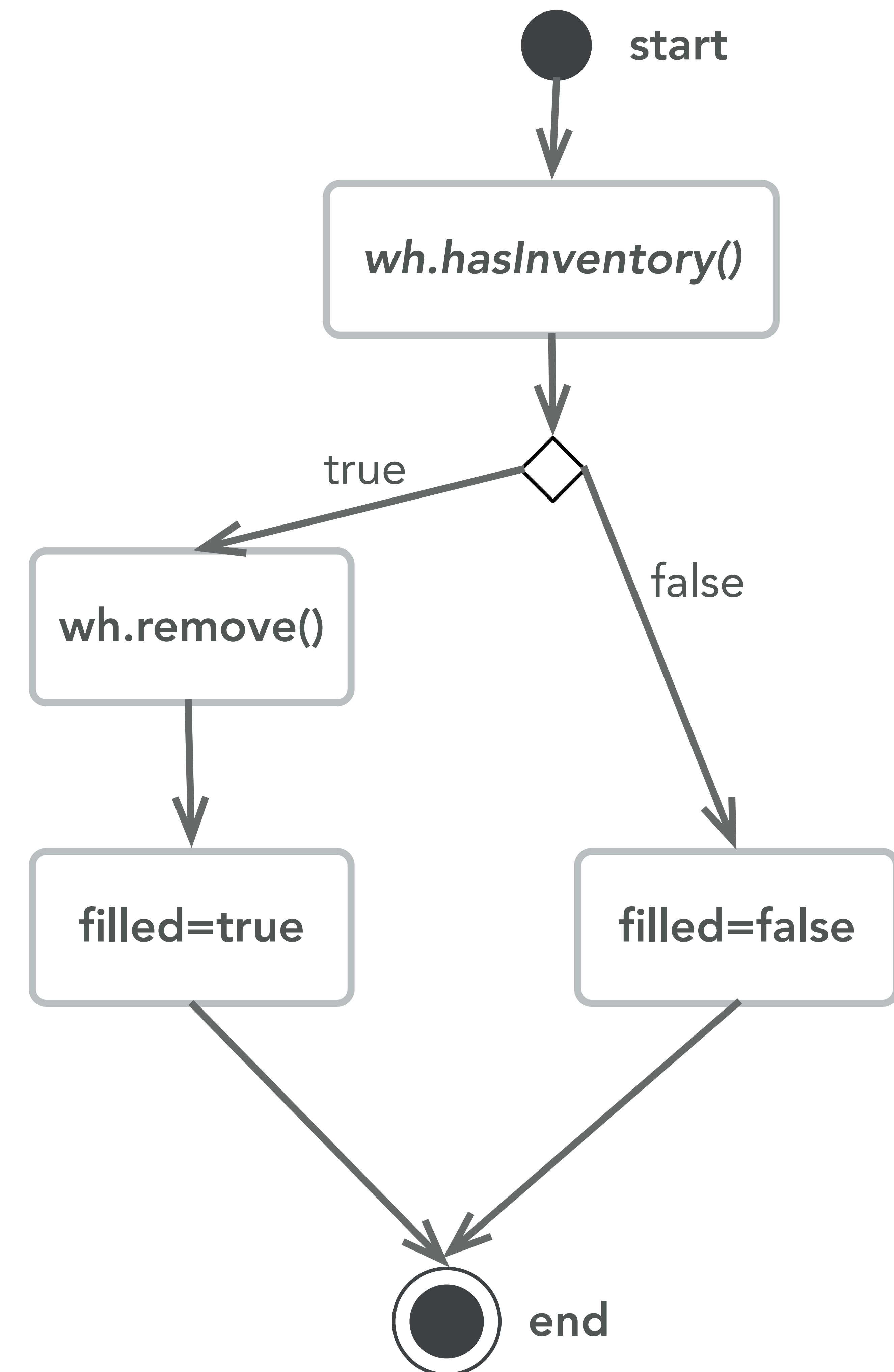
© Peter Sommerlad

Fowler's Wiskey-Store Example

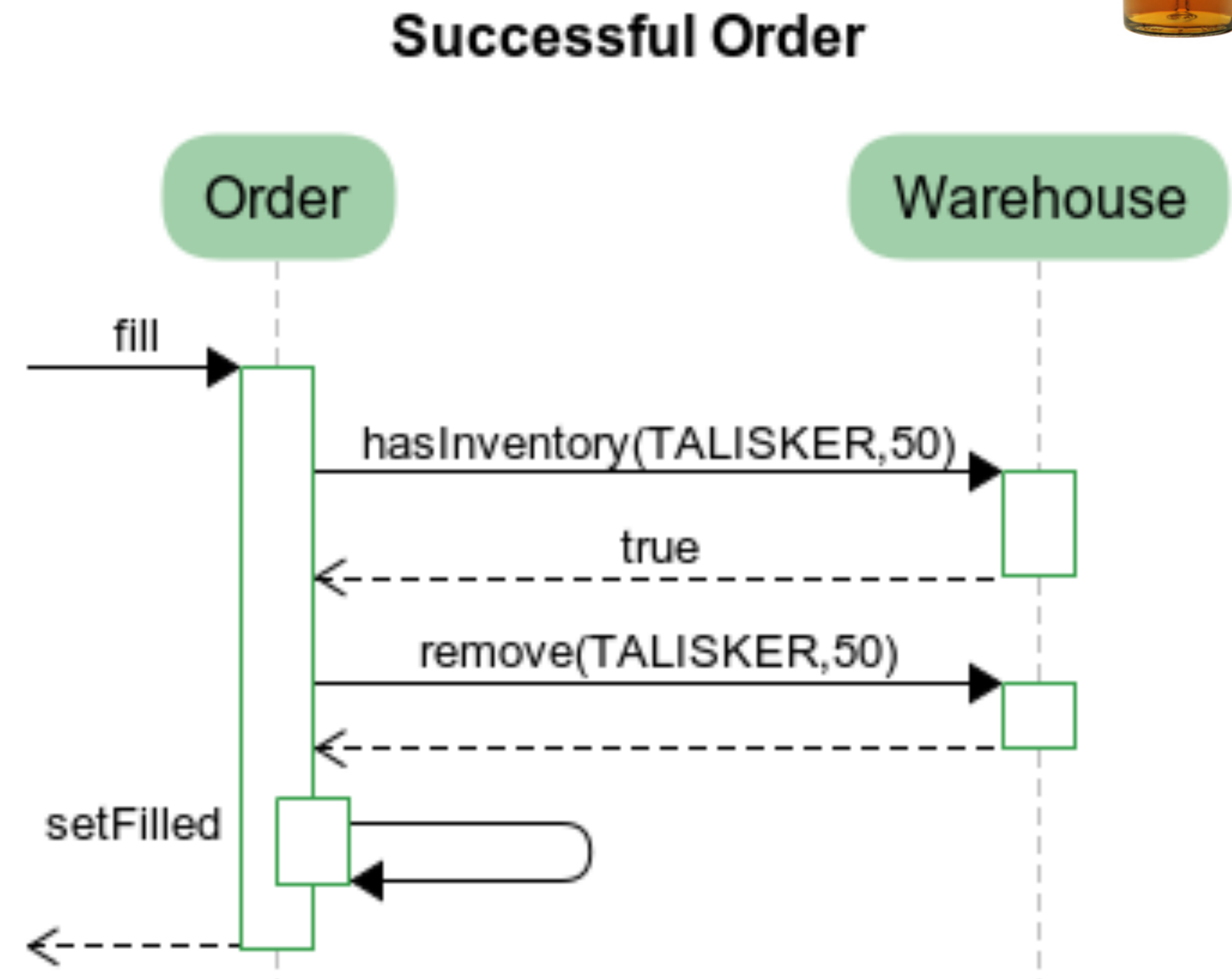
- Intentionally bad design…?

© Peter Sommerlad

Intended behavior of Order object



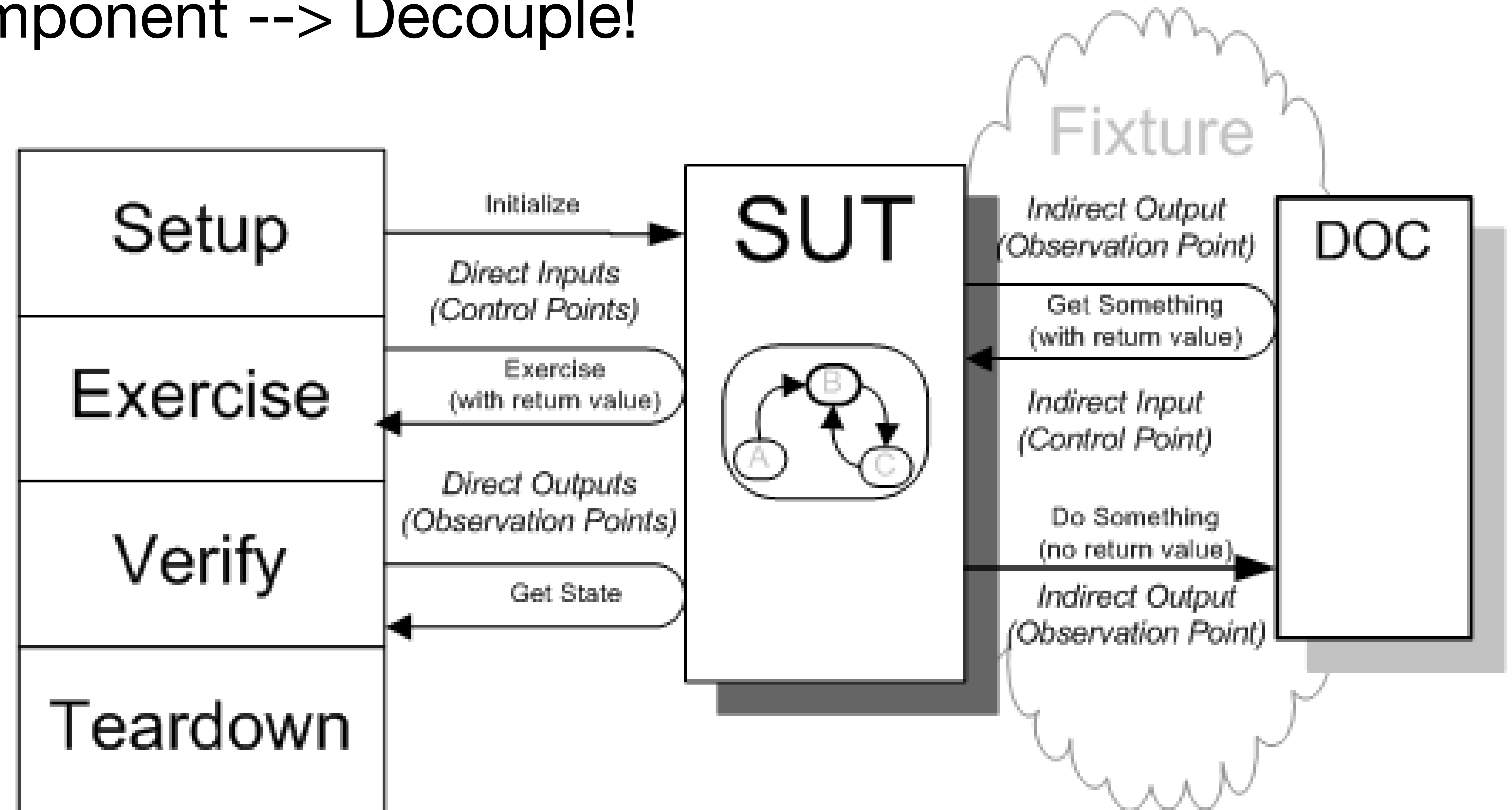
- Mocking is often used for stateful APIs
- verify usage of Warehouse (DOC) by Order (SUT)
- check call sequence from Order (SUT)



© Peter Sommerlad

Tests for stuff with dependencies

- DOC - Dependent other Component --> Decouple!
- SUT - System under Test



- source: xunitpatterns.com

© Peter Sommerlad

Why you need to stub/mock?

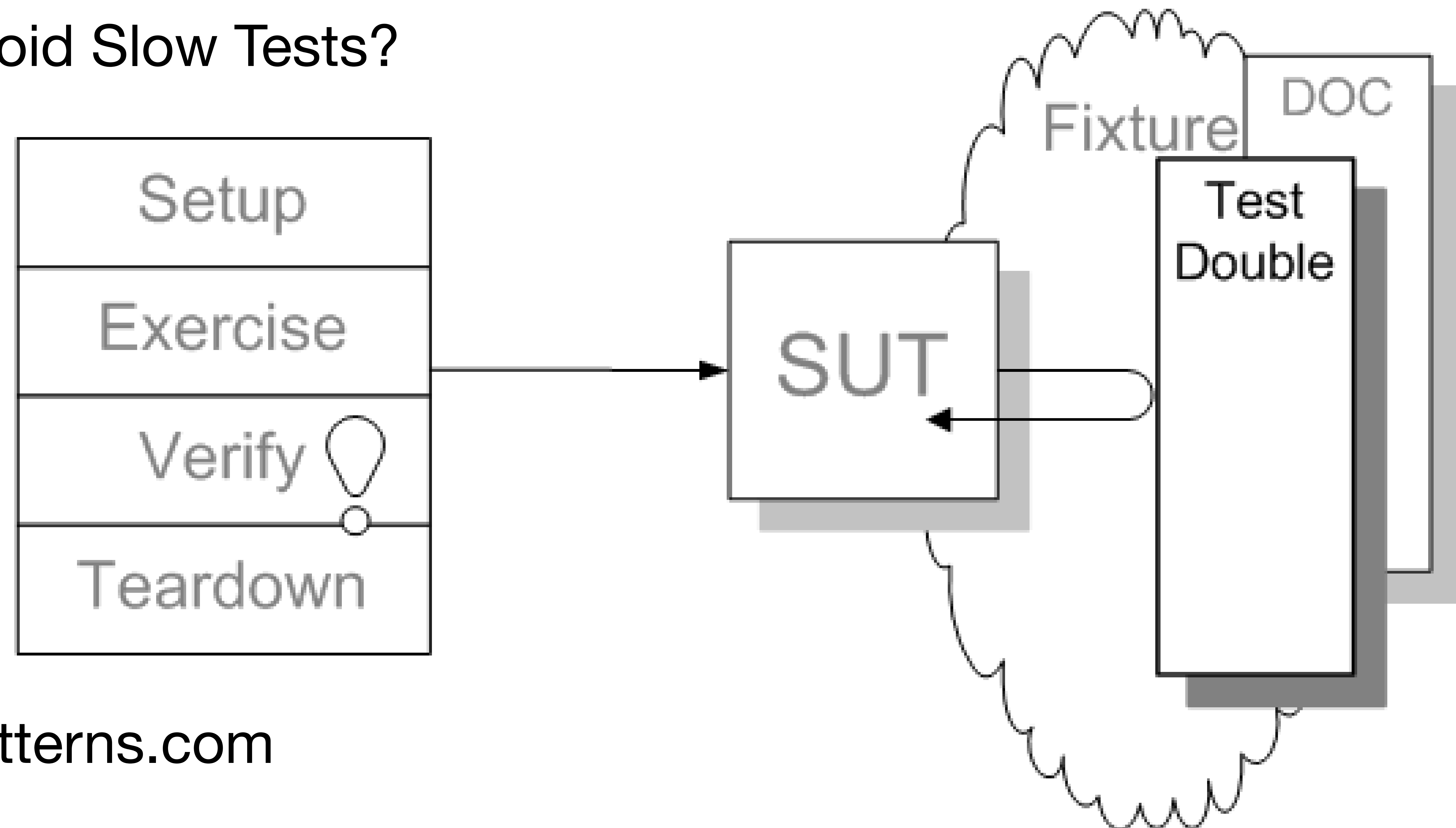
- **The real object has nondeterministic behavior.**
  - it produces unpredictable results, like a stock-market quote feed.
- **The real object is difficult to set up.**
- **The real object has hard to trigger behavior.**
  - for example, a network error or out-of-memory condition
- **The real object is slow.**
  - like real hardware, e.g., a motor movement
- **The real object has or is a user interface.**
  - or the user herself
- **The test needs to ask the real object about how it was used.**
  - for example, a test might need to confirm that a callback function was actually called.
- **The real object does not yet exist.**
  - a common problem when interfacing with other teams or new hardware systems

source: Pragmatic Unit Testing

© Peter Sommerlad

Test double patterns

- How can we verify logic independently when code it depends on is unusable?
- How can we avoid Slow Tests?

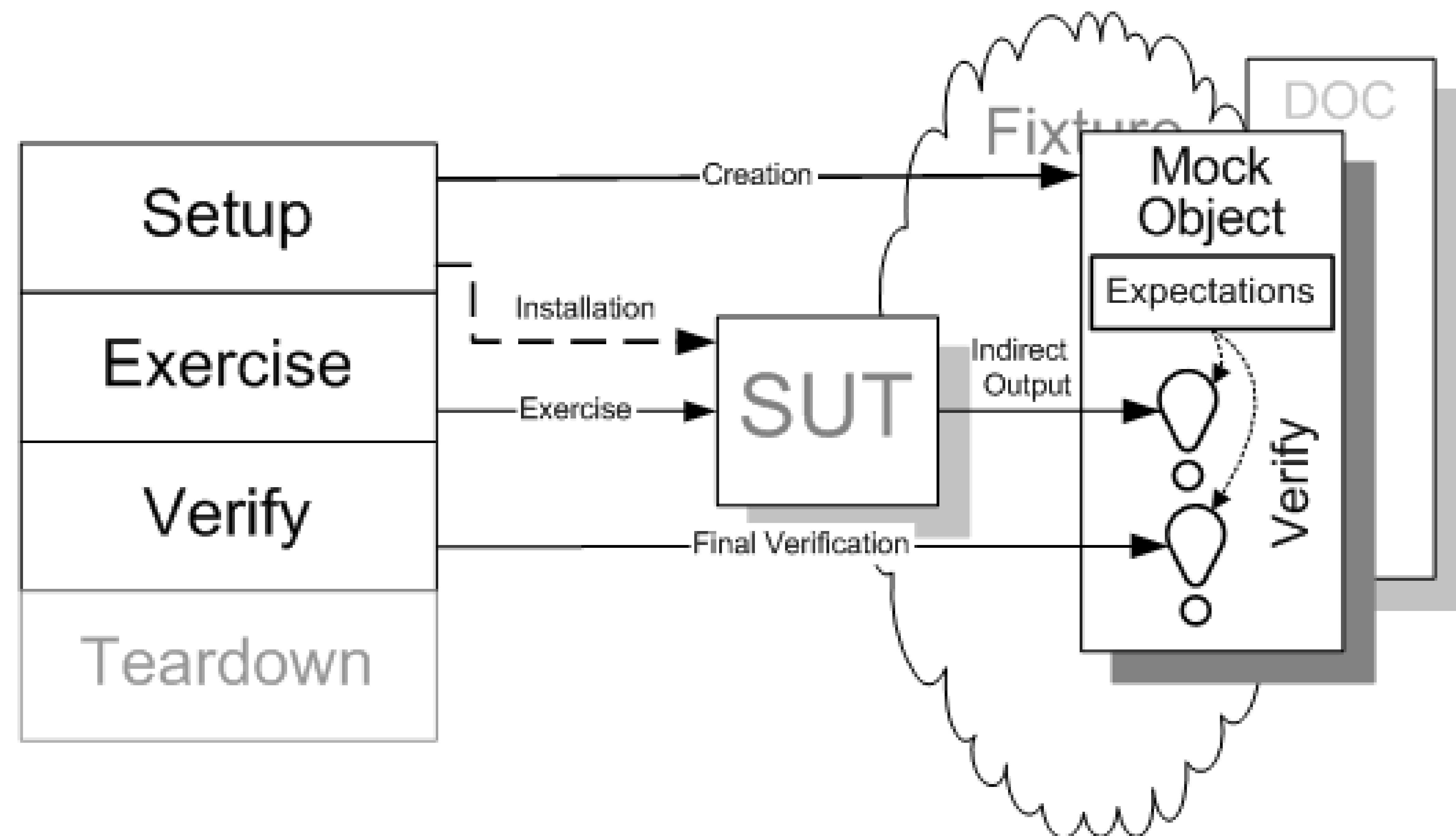


- source: xunitpatterns.com

© Peter Sommerlad

Mock object patterns

- How do we implement **Behavior Verification** for indirect outputs of the SUT?
- How can we verify logic independently when it depends on indirect inputs from other software components?
- source: xunitpatterns.com

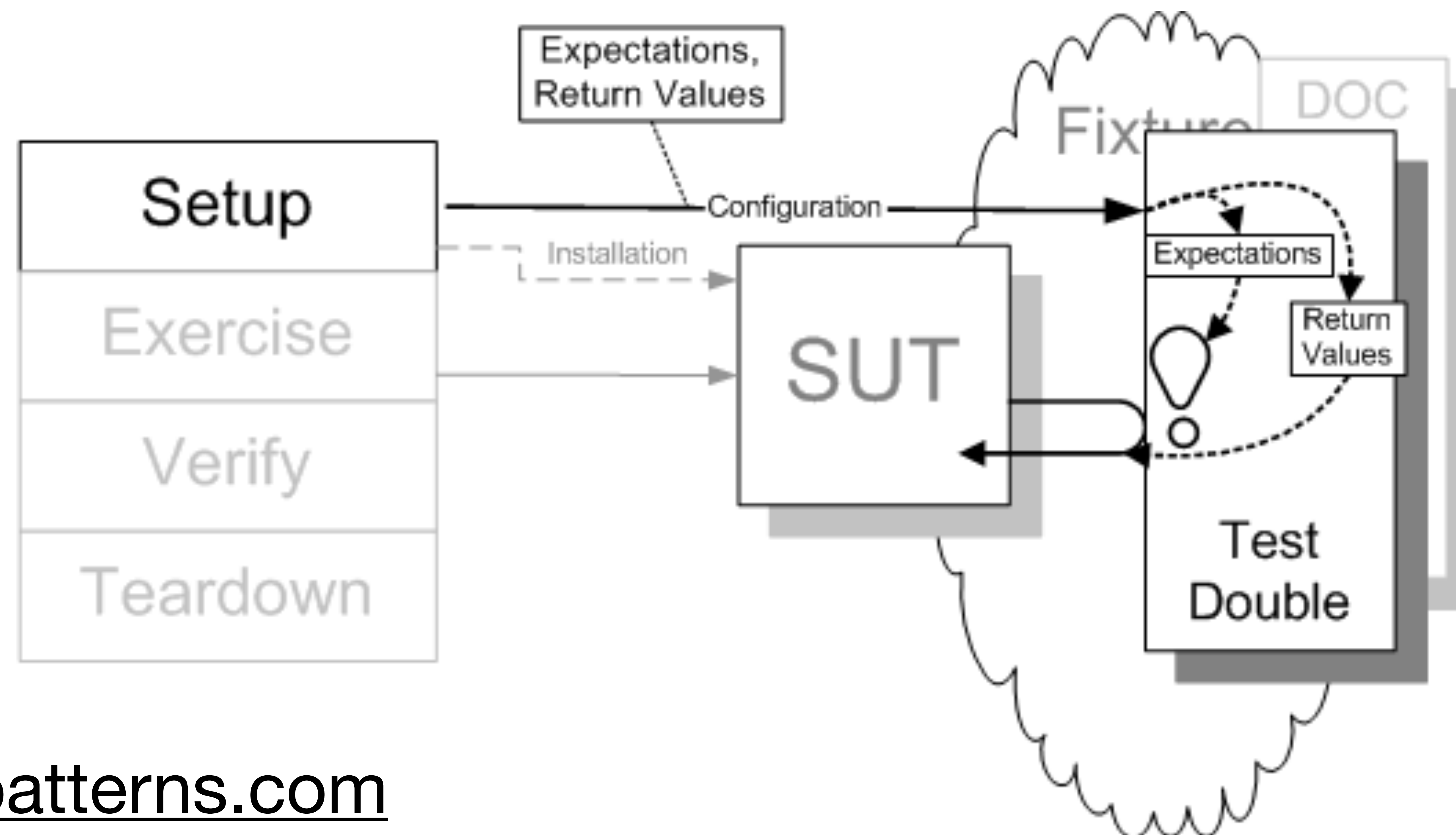


© Peter Sommerlad

Configurable Test Double

IMHO: overused idea in Mocking Frameworks

- How do we tell a Test Double what to return or expect?
- Configure a reusable Test Double with the values to be returned or verified during the fixture setup phase of a test.



Java at the time of JMock's invention didn't have Lambdas or a simple way to specify tiny pieces of behavior, everything needed to be a class (even an inner)

- source: xunitpatterns.com

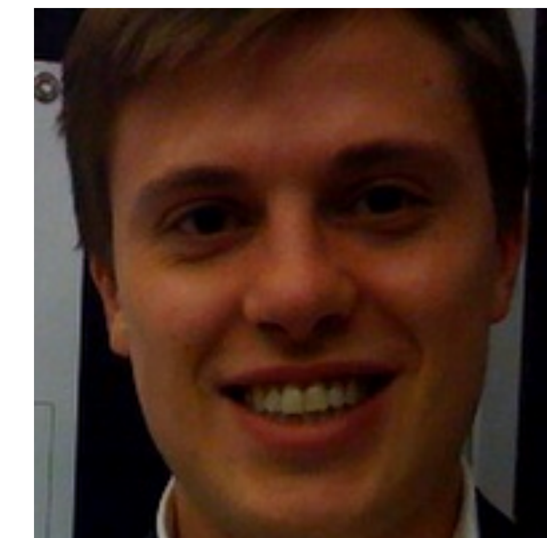
© Peter Sommerlad

What is in a Mocking Framework?

- **Dependency Injection** (possibly separate, might require refactoring)
  - replace DOC by Test Double - “**Introduce Seam**” - WEWLC
- **Fake** DOC’s function results for SUT
- **Trace** calls and arguments of SUT on DOC’s Test Double
- **Match** sequence of calls and argument values made on DOC from SUT

© Peter Sommerlad

C++ Mocking Framework Examples

GMock	Trompelœil	Mockator/Cevelop
		
Zhanyong Wan	Björn Fahller	Michael Rüegg

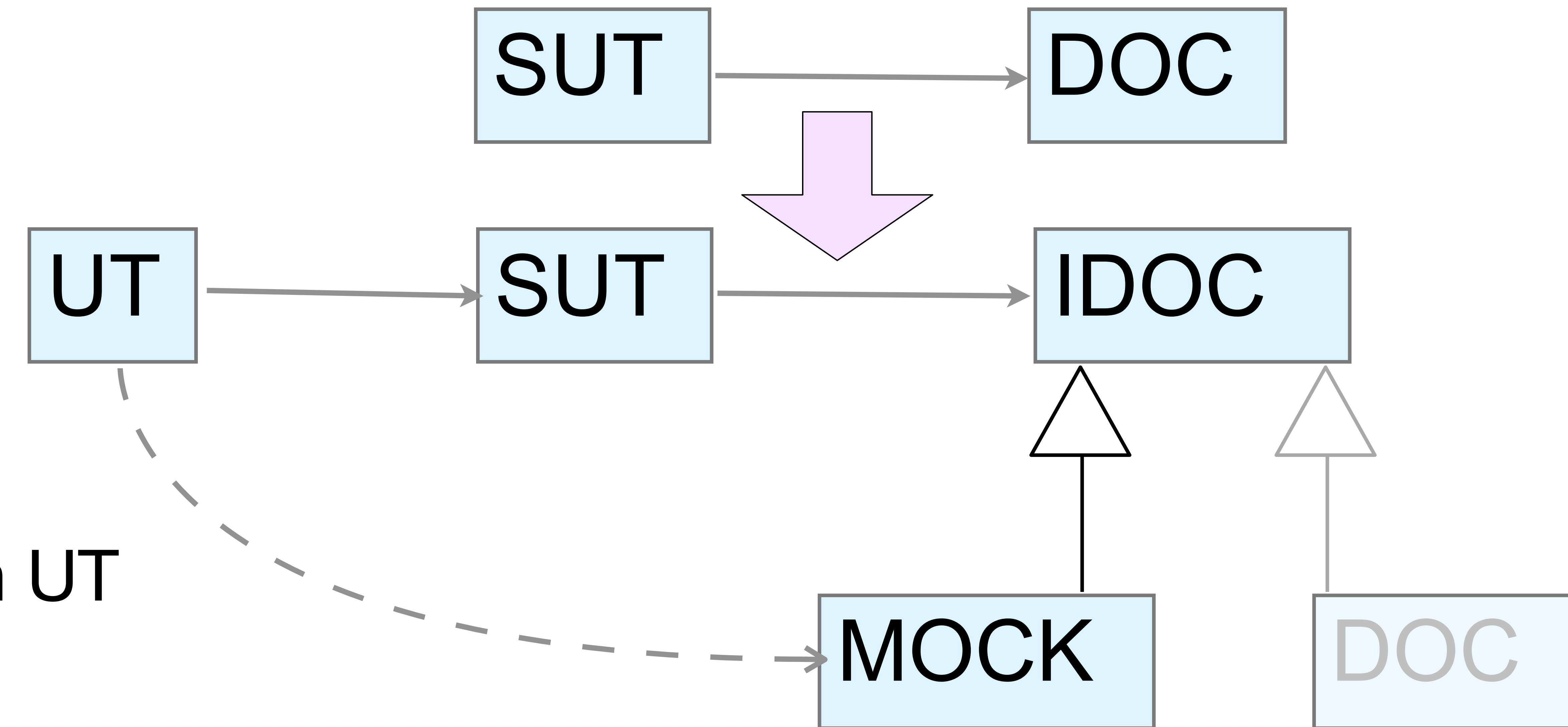
Introducing Seams

- *A seam is a place where you can alter behavior in your program without editing in that place.*
  - Michael Feathers, *Working Effectively with Legacy Code*
- **Object Seam**: Introduce Interface (virtual with inheritance)
  - pass DOC/Test Double as a (constructor) argument by reference
- **Compile Seam**: Extract Template Parameter
  - make DOC default template argument, pass Test Double in tests
- **Linker Seam**, **Preprocessor Seam** other possibilities of “last resort” or for C-functions

© Peter Sommerlad

Introducing Mockability

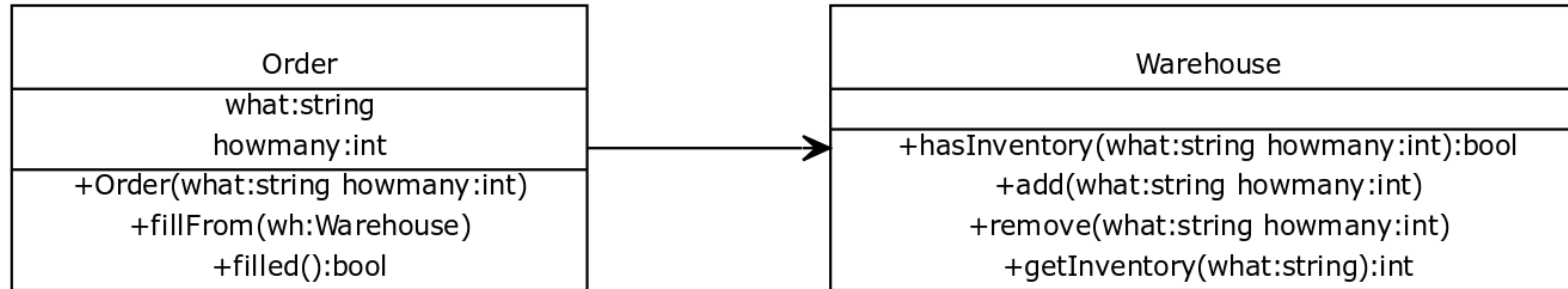
- classic inheritance based mocking
- extract interface for DOC -> IDOC
- make SUT use IDOC
- create MOCK implementing IDOC and use it in UT



- in C++ this means often overhead for DOC (virtual functions)!

© Peter Sommerlad

Introduce Mockability by Extract Interface Refactoring



UnitTest

Order
what:string
howmany:int
+Order(what:string howmany:int)
+fillFrom(wh:Warehouse)
+filled():bool

IWarehouse
+hasInventory(what:string howmany:int):bool
+add(what:string howmany:int)
+remove(what:string howmany:int)
+getInventory(what:string):int

Warehouse
+hasInventory(what:string howmany:int):bool
+add(what:string howmany:int)
+remove(what:string howmany:int)
+getInventory(what:string):int

MockWarehouse
+hasInventory(what:string howmany:int):bool
+add(what:string howmany:int)
+remove(what:string howmany:int)
+getInventory(what:string):int

© Peter Sommerlad

Cevelop : Extract Interface

Your C++ code deserves it

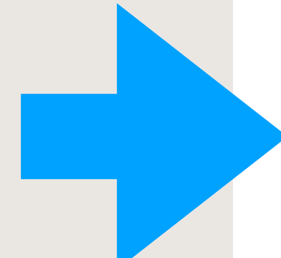
```cpp
struct Warehouse {
   int getInventory(std::string const& s) const {
      auto const it = wh.find(s);
      return (it != wh.end())?it->second:0;
   }

   void remove(std::string const & s, int i) {
      wh[s]-=i;
   }

   void add(std::string const & s, int i) {
      wh[s]+=i;
   }

   bool hasInventory(const std::string& s, int i) const {
      return getInventory(s)>=i;
   }
   std::map<std::string,int> wh{};
};
struct Order {

   void fill(Warehouse& warehouse) {
//. . .
```



```cpp
struct WarehouseInterface {
   virtual ~WarehouseInterface() {
   }

   virtual int getInventory(std::string const & s) const = 0;
   virtual void remove(std::string const & s, int i) = 0;
   virtual void add(std::string const & s, int i) = 0;
   virtual bool hasInventory(std::string const & s, int i) const = 0;
};
struct Warehouse: WarehouseInterface {
   int getInventory(std::string const & s) const {
//. . .
};
struct Order {

   void fill(WarehouseInterface& warehouse) {
//. . .
};
```

© Peter Sommerlad

Compile-seam with


```
void OrderFillFromWarehouse(){
    struct MockWarehouse {
    };
    MockWarehouse warehouse { };
    OrderT<MockWarehouse> order(TALISKER,50);
    order.fill(warehouse);
    ASSERT(not order.isFilled());
}
```

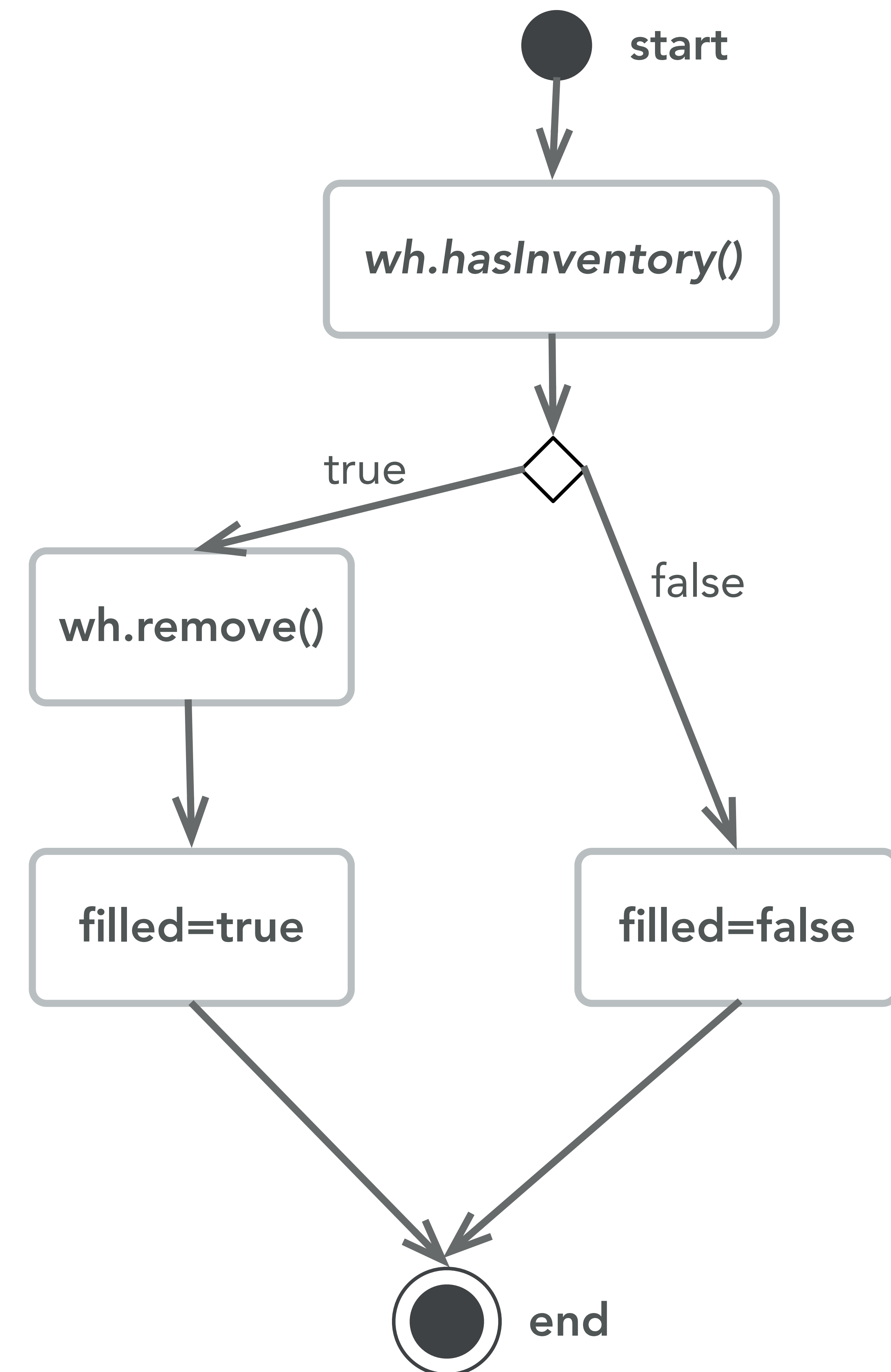
Necessary member function(s) not existing in class MockWarehouse

```
  OrderT<MockWarehouse> order(TALISKER,50);
  order.fill(warehouse);
  ASSERT(not order.isFilled());
}
```

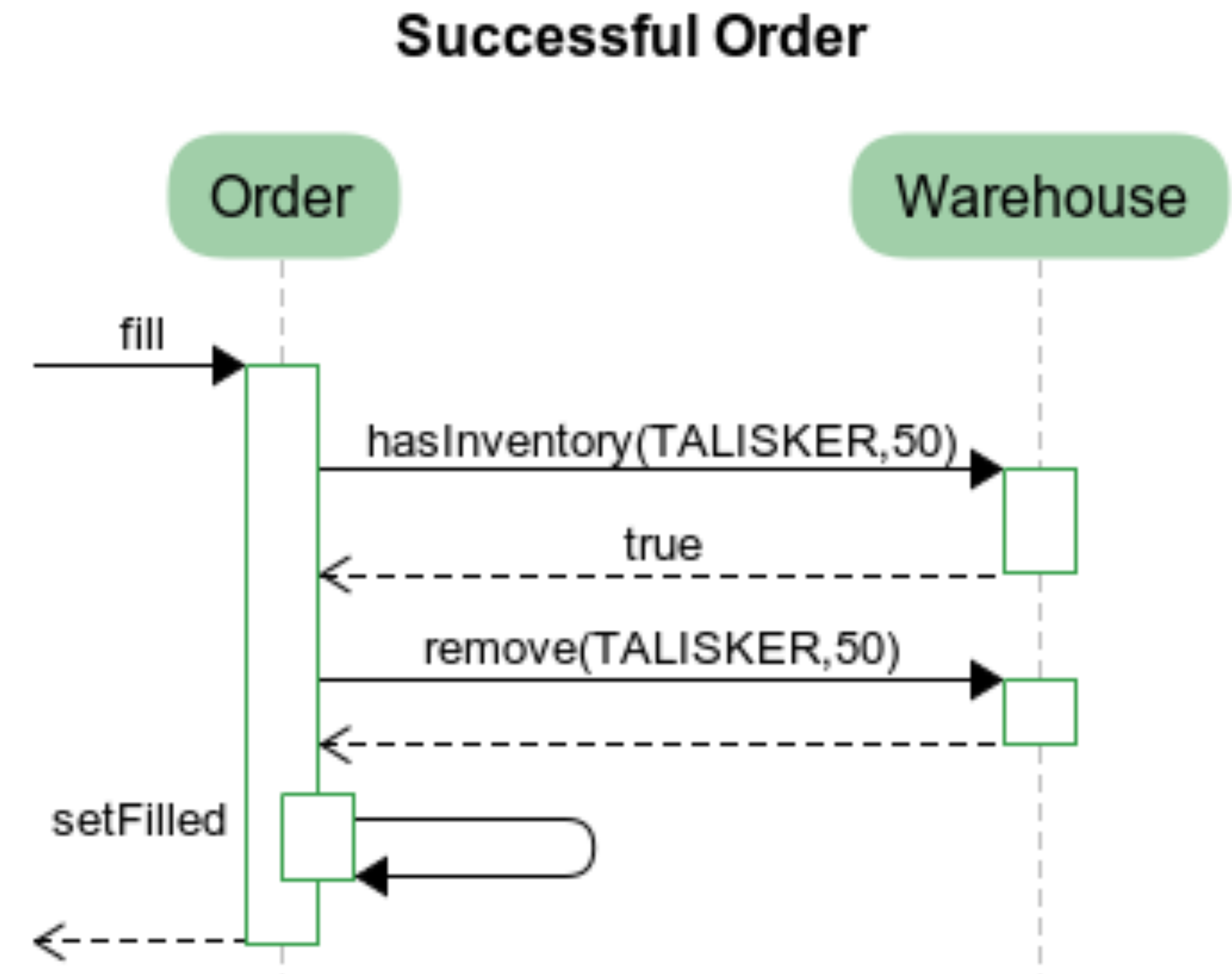
First: Extract Template Parameter Refactoring enables Seam

© Peter Sommerlad

Intended behavior of Order object



- How to provide behavior for a fake Warehouse?
- How do we test for intended usage of Warehouse?



© Peter Sommerlad

Fake Classes - helps


```
struct WarehouseInterface {
   virtual int getInventory(std::string const & s) const = 0;
   virtual void remove(std::string const & s, int i) = 0;
   virtual void add(std::string const & s, int i) = 0;
   virtual bool hasInventory(std::string const & s, int i)
const = 0;
};
```

```
struct EmptyWarehouse :WarehouseInterface{
};

void Or
   Ord
   Emp
```

- Add missing member functions to class EmptyW
- Record calls by choosing function arguments in
- Record calls by choosing function order in class
- Rename in file
- Rename in workspace

4 member function(s) to implement:
add(const std::string&, int)
getInventory(const std::string&) const
hasInventory(const std::string&, int) const
remove(const std::string&, int)

```
void OrderFillFromWarehouse(){
   Order order(TALISKER,50);
   EmptyWarehouse warehouse{};
   order.fill(warehouse);
   ASSERT(not order.isFilled());
}
```

```
struct EmptyWarehouse :WarehouseInterface{
   void add(std::string const & s, int i) {
   }

   int getInventory(std::string const & s) const {
      return int { };
   }

   bool hasInventory(std::string const & s, int i) const {
      return bool { };
   }

   void remove(std::string const & s, int i) {
   }
};
```

```
void OrderFillFromWarehouse(){
   Order order(TALISKER,50);
   EmptyWarehouse warehouse{};
   order.fill(warehouse);
   ASSERT(not order.isFilled());
}
```

© Peter Sommerlad

Preparation for Mocking

```
struct WarehouseInterface {
  virtual int getInventory(std::string const & s) const = 0;
  virtual void remove(std::string const & s, int i) = 0;
  virtual void add(std::string const & s, int i) = 0;
  virtual bool hasInventory(std::string const & s, int i) const = 0;
};
```

Extract Interface required for all mocking frameworks

```
struct MockWarehouse: WarehouseInterface {
public:
  MOCK_CONST_METHOD1(getInventory, int(std::string const&));
  MOCK_METHOD2(remove, void(std::string const&,int));
  MOCK_METHOD2(add, void(std::string const&,int));
  MOCK_CONST_METHOD2(hasInventory, bool(std::string const&,int));
};
```

GMock

MACROS 🤮

CONST

```
struct MockWarehouse: WarehouseInterface {
  MAKE_CONST_MOCK1(getInventory, int(std::string const&),override);
  MAKE_MOCK2(remove, void(std::string const&,int),override);
  MAKE_MOCK2(add, void(std::string const&,int),override);
  MAKE_CONST_MOCK2(hasInventory,bool(std::string const &,int),override);
};
```

Trompelœil

© Peter Sommerlad

Faking with Mocking Frameworks

```
  bool hasInventory(std::string const & s, int i) const {
    return false;
  }
//...
void OrderFillFromWarehouse(){
  Order order(TALISKER,50);
  EmptyWarehouse warehouse{};
  order.fill(warehouse);
  ASSERT(not order.isFilled());
}
```

DIY/ Mockator

```
TEST(OrderTest, EmptyWarehouse)
{
  MockWarehouse warehouse{};
  Order order{TALISKER,50};
  EXPECT_CALL(warehouse,hasInventory(TALISKER,50)).WillOnce(Return(false));
  //ON_CALL(warehouse,hasInventory(TALISKER,50)).WillByDefault(Return(false));
  order.fill(warehouse);
  ASSERT_FALSE(order.isFilled());
}
```

GMock/GTest

ON_CALL causes run-time warning but would be the right thing to do. You need to use NiceMock<MockWarehouse> to compensate that.

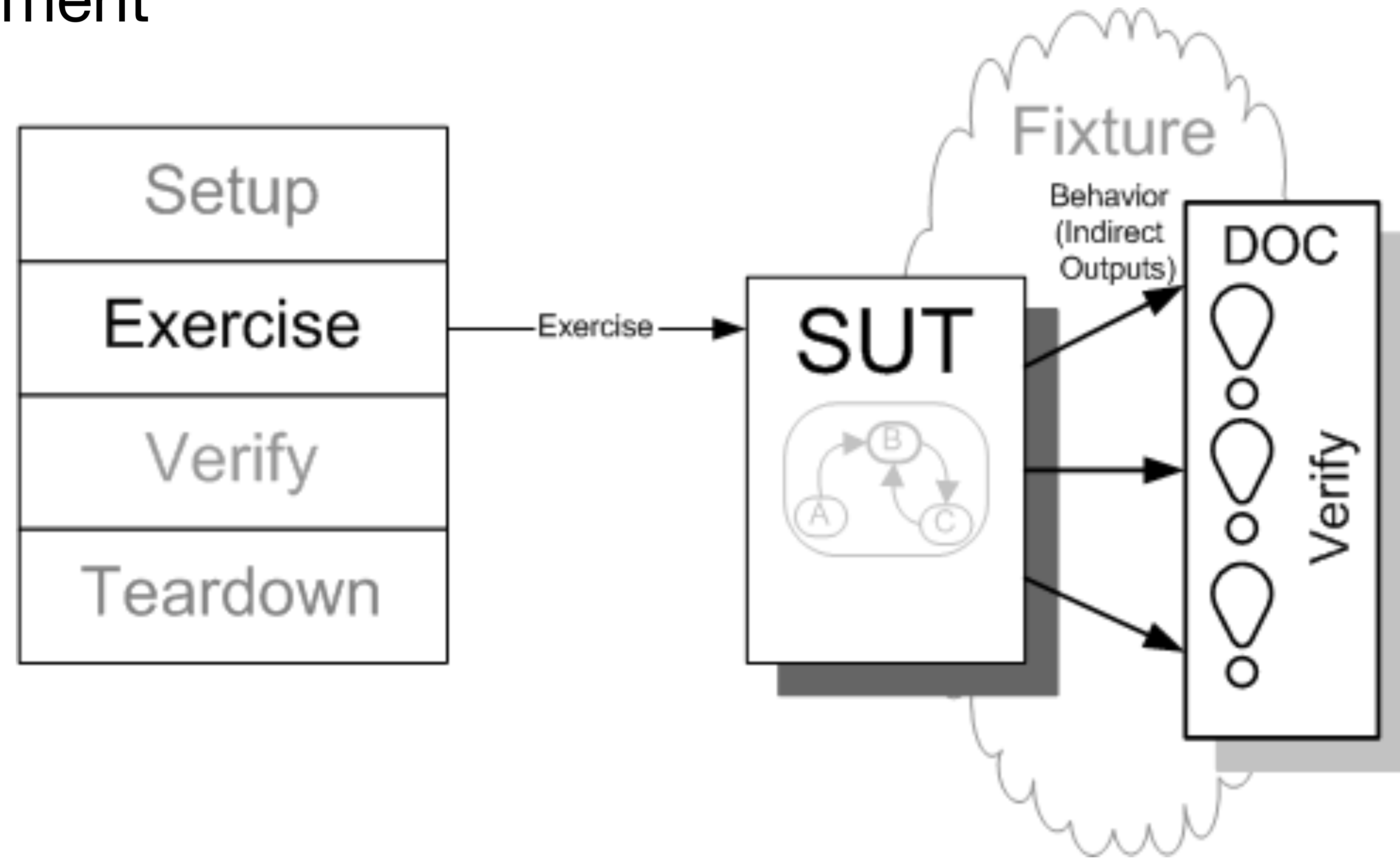
```
void testMockingWithTrompeloeil(){
  MockWarehouse wh;
  //REQUIRE_CALL(wh,hasInventory(TALISKER,50)).RETURN(false).TIMES(1);
  ALLOW_CALL(wh,hasInventory(TALISKER,50)).RETURN(false);
  Order order(TALISKER,50);
  order.fill(wh);
  ASSERT(not order.isFilled());
}
```

Trompelœil

© Peter Sommerlad

Behavior Verification

- “need-driven” outside-in development
- White-box test
- DOC not-yet-existent
- Fragile tests
- Over-specification
- “Hard Tests” hinders Refactoring



© Peter Sommerlad

Sequence expectations

```
TEST(OrderTest, FilledWarehouse)
{
   MockWarehouse warehouse{};
   InSequence s{};
   EXPECT_CALL(warehouse,hasInventory(TALISKER,50)).WillOnce(Return(true));
   EXPECT_CALL(warehouse,remove(TALISKER,50));
   Order order{TALISKER,50};
   order.fill(warehouse);
   ASSERT_TRUE(order.isFilled());
}
```

GMock

Implicit expectation check on scope exit

```
void testFulledOrderWithTrompeloeil(){
   MockWarehouse wh{};
   trompeloeil::sequence seq{};
   REQUIRE_CALL(wh,hasInventory(TALISKER,50)).RETURN(true).TIMES(1).IN_SEQUENCE(seq);
   REQUIRE_CALL(wh,remove(TALISKER,50)).TIMES(1).IN_SEQUENCE(seq);
   Order order(TALISKER,50);
   order.fill(wh);
   ASSERT(order.isFilled());
}
```

Trompelœil

Implicit expectation check on scope exit

© Peter Sommerlad

Problem: DSLs for specifying behavior

obj : Mock

Ceci n'est pas un objet

Trompeloeil cheat sheet for implementing mock functions and placing expectations on them.

Mock implement member functions.

non-const member function	*const member function*
`MAKE_MOCKn(name, sig{, spec})`	`MAKE_CONST_MOCKn(name, sig{, spec})`

Place expectations. *Matching expectations are searched from youngest to oldest. Everything is illegal by default.*

Anonymous local object	*std::unique_ptr<expectation>*
`REQUIRE_CALL(obj, func(params))`	`NAMED_REQUIRE_CALL(obj, func(params))`
`ALLOW_CALL(obj, func(params))`	`NAMED_ALLOW_CALL(obj, func(params))`
`FORBID_CALL(obj, func(params))`	`NAMED_FORBID_CALL(obj, func(params))`

Refine expectations.

When to match

- `.IN_SEQUENCE(s...)` ← Impose an ordering relation between expectations by using **`sequence`** objects
- `.TIMES(min, max = min)` ← Define how many times an expectation must match. Default is 1. Conveniency arguments are **`AT_MOST(x)`** and **`AT_LEAST(x)`**

Parameters are _1 .. _15

Local objects are const copies		*Local objects are non-const references*
`.WITH(condition)`	← when to match →	`.LR_WITH(condition)`
`.SIDE_EFFECT(statement)`	← What to do when matching →	`.LR_SIDE_EFFECT(statement)`
`.RETURN(expression)`	← →	`.LR_RETURN(expression)`
`.THROW(expression)`	← →	`.LR_THROW(expression)`

obj : Mock

Ceci n'est pas un objet

Trompeloeil cheat sheet for matchers and object life time management.

Matchers. Substitute for values in parameter list of expectations.

Any type allowing op				*Disambiguated type*
`_`	any value			`ANY(type)`
`eq(mark)`	value	==	mark	`eq<type>(mark)`
`ne(mark)`	value	!=	mark	`ne<type>(mark)`
`lt(mark)`	value	<	mark	`lt<type>(mark)`
`le(mark)`	value	<=	mark	`le<type>(mark)`
`gt(mark)`	value	>	mark	`gt<type>(mark)`
`ge(mark)`	value	>=	mark	`ge<type>(mark)`
`re(mark, ...)`	match regular expression /mark/			`re<type>(mark, ...)`

Use **`operator*`** to dereference pointers. E.g. **`*ne(mark)`** means parameter is pointer (like) and **`*parameter != mark`**
Use **`operator!`** to negate matchers. E.g. **`!re(mark)`** means not matching regular expression **`/mark/`**

Object life time management

`auto obj = new deathwatched<my_mock_type>(params);`
`*obj` destruction only allowed when explicitly required. Inherits from **`my_mock_type`**

Anonymous local object	*std::unique_ptr<expectation>*
`REQUIRE_DESTRUCTION(*obj)`	**`NAMED_REQUIRE_DESTRUCTION(*obj)`**

When to match

- `.IN_SEQUENCE(s...)` ← Impose an ordering relation between expectations by using **`sequence`** objects

© Peter Sommerlad

Kent Beck:

“Do the simplest thing that could possibly work”

When you don’t know what to do.

© Peter Sommerlad

Ward Cunningham:

“Kent,
Do the simplest thing that could possibly work”

When you don’t know what to do.

© Peter Sommerlad

Simpler Mocking with Mockator

```
MockWarehouse warehouse { };
OrderT<MockWarehouse> order(TALISKER,50);
order.fill(warehouse);
ASSERT( order.isFilled());
calls expectedMockWarehouse{
   call("MockWarehouse()"),
   call("hasInventory(const std::string&, int) const", TALISKER,50),
   call("remove(const std::string&, int) const", TALISKER,50)
};
ASSERT_EQUAL(expectedMockWarehouse, allCalls[1]);
```

- NO #define MACROS for users (almost)
- Introduce Seam refactorings
- Generate **regular C++** code for Seam by IDE (Cevelop)
- Generate **regular C++** code for tracing calls by IDE (Cevelop) on demand
- Use vector<call> for tracing calls (call ~ std::string)
- Use C++ and std::regex for matching, if needed

see videos at http://mockator.com

© Peter Sommerlad

Sequencing with Mockator

```
void OrderFillFromWarehouse(){
    INIT_MOCKATOR();
    static std::vector<calls> allCalls(1);
    struct MockWarehouse {
        size_t const mock_id;

        MockWarehouse()
        :mock_id(reserveNextCallId(allCalls))
        {
            allCalls[mock_id].push_back(call("MockWarehouse()"));
        }

        bool hasInventory(std::string const & what, int howmany) const {
            allCalls[mock_id].push_back(call("hasInventory(const std::string&, int) const", what, howmany));
            return bool();
        }

        void remove(std::string const & what, int howmany) const {
            allCalls[mock_id].push_back(call("remove(const std::string&, int) const", what, howmany));
        }
    };
    MockWarehouse warehouse { };
    OrderT<MockWarehouse> order(TALISKER,50);
    order.fill(warehouse);
    ASSERT(not order.isFilled());
    calls expectedMockWarehouse{
        call("MockWarehouse()"),
        call("hasInventory(const std::string&, int) const", TALISKER,50),
        call("remove(const std::string&, int) const", TALISKER,50)
    };
    ASSERT_EQUAL(expectedMockWarehouse, allCalls[1]);
}
```

All of MockWarehouse is generated by Cevelop

Result Comparison

Expected:	Actual:
[	[
MockWarehouse()	MockWarehouse()
hasInventory(const std::string&, int) const,Talisker,50	hasInventory(const std::string&, int) const,Talisker,50
remove(const std::string&, int) const,Talisker,50	]
]	

OK

© Peter Sommerlad

Mocking Frameworks (in general)

- Most Designs stems from early JMock/EasyMock (as GMock)
  - Java Mocking Frameworks
  - Reflection, only classes and objects, dynamic polymorphism,
    NO lambdas
- Subclassing with method implementation through reflection (Java-ish)
- no lambdas: dynamic construction of behavior, matching via reflection
  - domain-specific language (DSL) to specify behavior, not plain code

© Peter Sommerlad

C++ Mocking Frameworks Problems

- Design tend to follow JMock/EasyMock - keep Java Design- lack C++ strengths
- No useful reflection in C++: Macros for generating names, defining function stubs
- Subclassing, virtual member functions, sometimes fiddling via undefined behavior or low-level machine-specific ABI, e.g., replacing vtable entries (Hippomocks)
- DSL to specify (expected) behavior, not plain code, MACRO magic
  - with implicit matching of actual behavior vs expected in destructors
  - fragile with refactoring, hard to reuse across tests, bloated tests

```
EXPECT_CALL(turtle, GetY())
    .WillOnce(Return(100))
    .WillOnce(Return(200))
    .WillRepeatedly(Return(300));
```

© Peter Sommerlad

Too much Mocking is bad

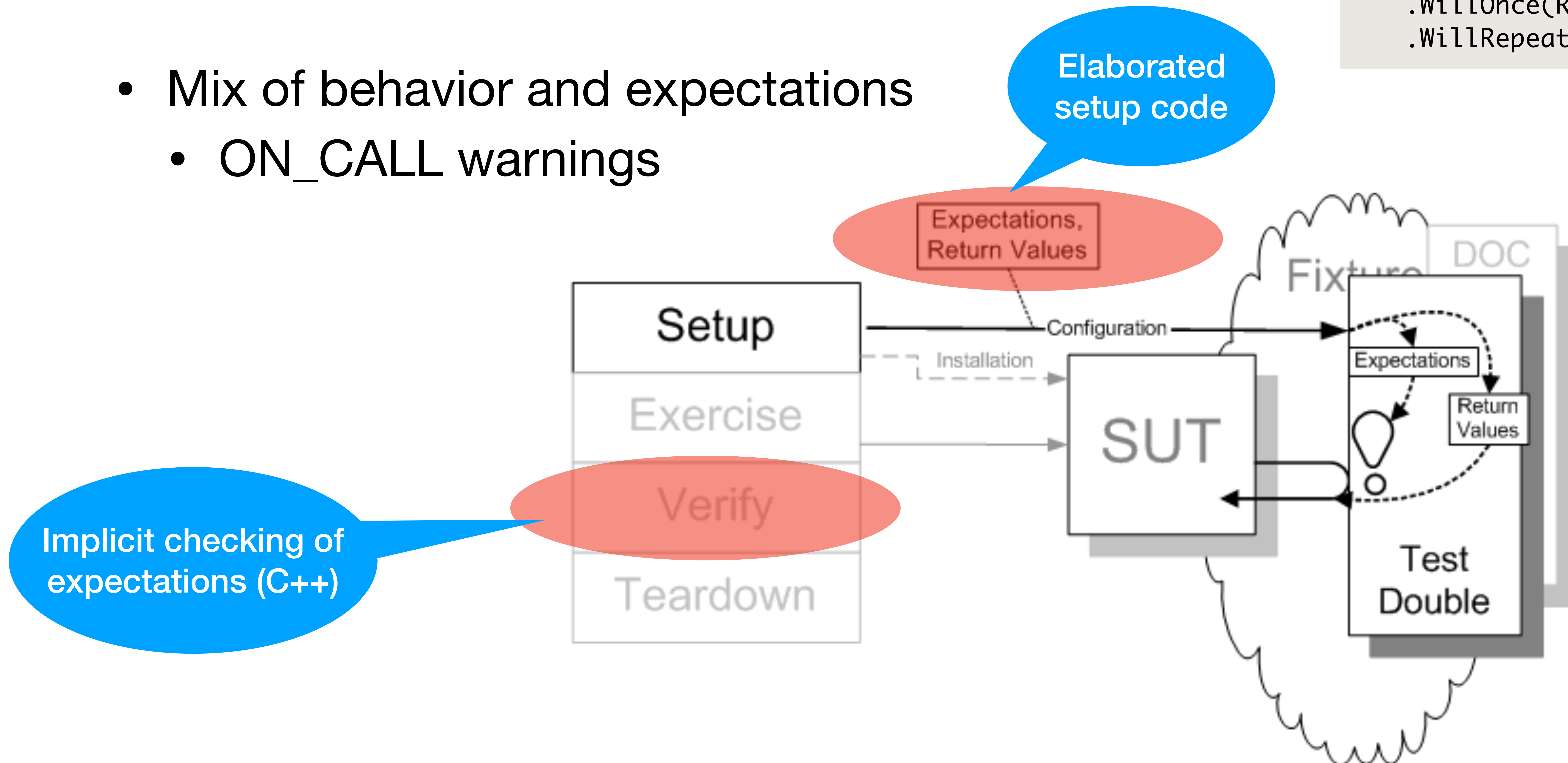
- **White box tests couple** Tests, SUT and DOC tight
  - hard to refactor for tools and manually, design flexibility is lost
- **Fosters stateful APIs**, especially when used with TDD of SUT and DOC
  - setWiggle(Wiggle), setWaggle(Waggle), WiggleTheWaggle() in the mocked object, called in sequence from SUT
- Tendency for single/no Parameter Functions and required sequence of when being called
  - Uncle Bob ("Clean Code") misunderstood: favor "niladic" functions
  - he says: "avoid temporal coupling"

© Peter Sommerlad

Problem: configurable test double

- DSL leads to complicated specification
- Mix of behavior and expectations
  - ON_CALL warnings

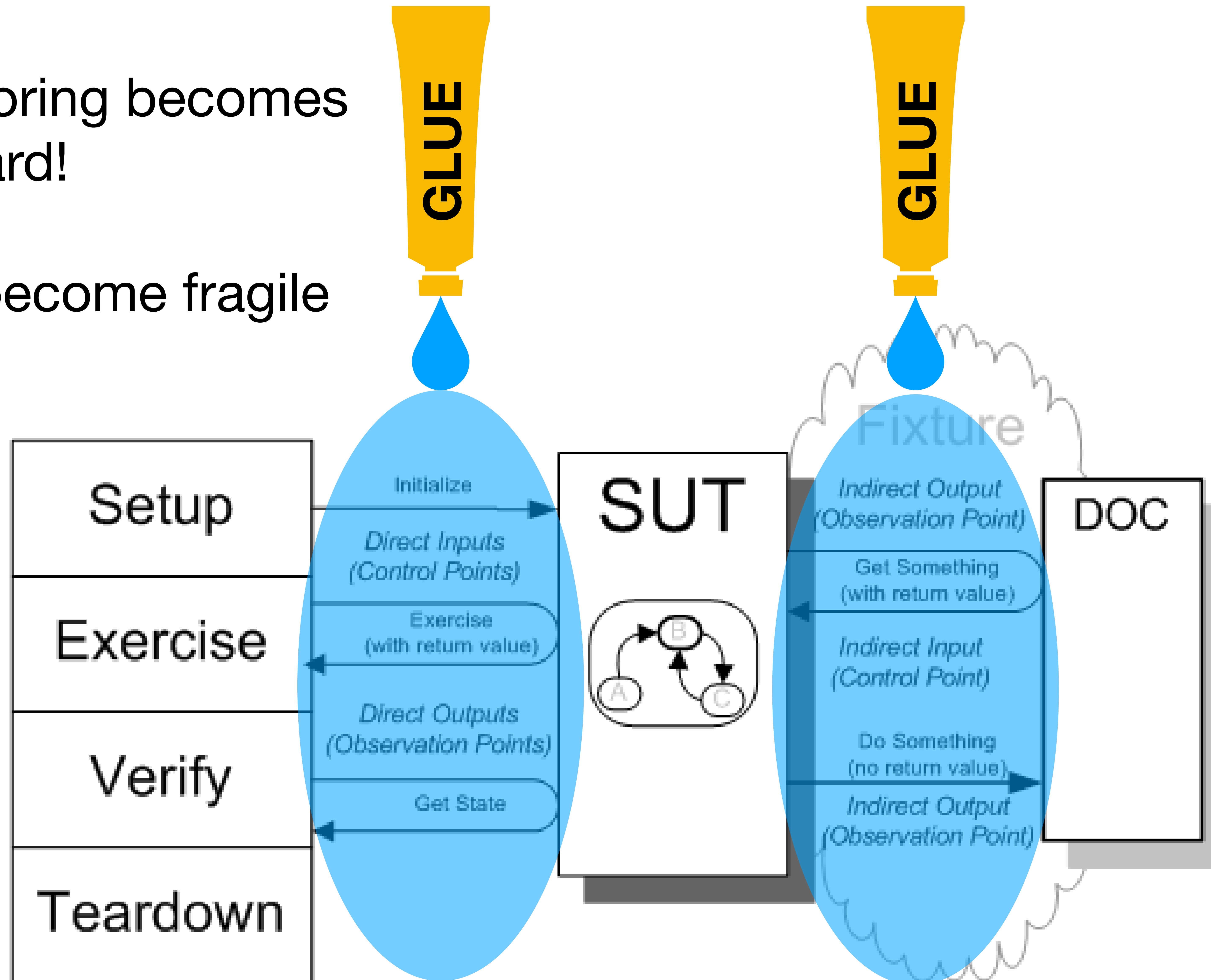
```
EXPECT_CALL(turtle, GetY())
    .WillOnce(Return(100))
    .WillOnce(Return(200))
    .WillRepeatedly(Return(300));
```



© Peter Sommerlad

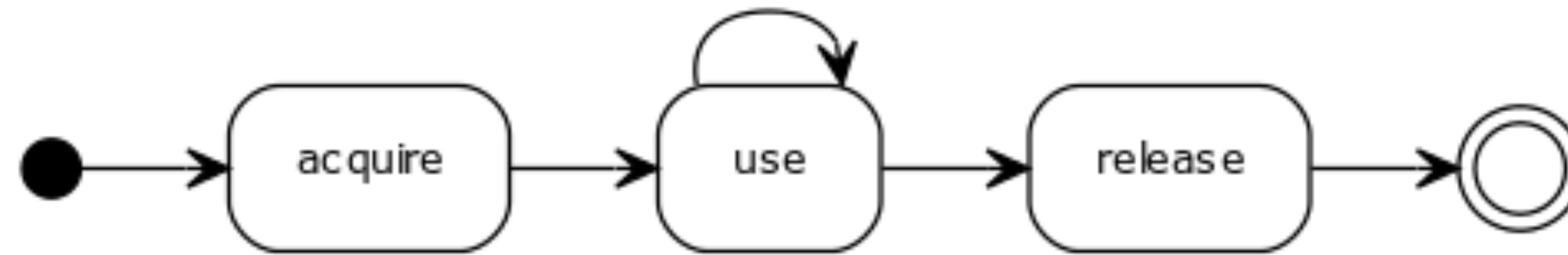
Danger: tight coupling

- Refactoring becomes very hard!
- Tests become fragile



© Peter Sommerlad

Stateful APIs are bad?!



- acquiring a resource, use it, release it - stateful?
- if RAII => OK: require, use, release (in dtor)
  - open(“hello.txt”), write/read, close
- else => NOK: can forget to release, two-phase init 💩
- Other more elaborated stateful APIs are (often) bad! == hard to test
  - socket, bind, listen, accept,…

© Peter Sommerlad

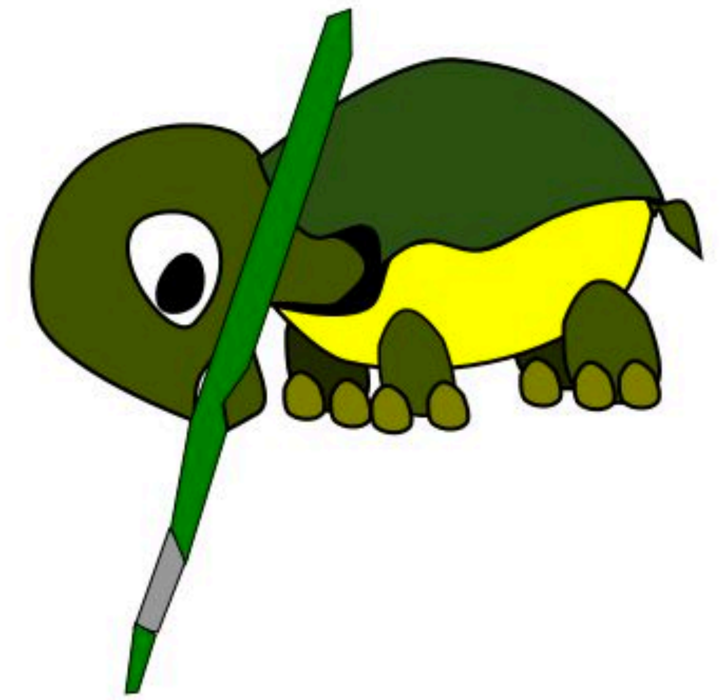
Ex. Bad Stateful APIs: sockets

- UNIX classic APIs for resources: file: open - read/write/… - close
  - single step init - operate - finish
- BSD sockets put things together with optional functionality and require multi-stage init (and thus suck):
  - socket, bind, listen, accept - read/write/… - close
  - socket, bind (opt), connect - read/write/… - close
- Many wrappers in different languages keep socket initialization multi-step
- Do not follow that lead for your own libraries/code - using correctly is hard!

© Peter Sommerlad

Stateful APIs are bad? Really?

- many graphics library are stateful
  - e.g. Turtle graphics, Cairo
- long sequences of “setters”, then action
- multi-step initialization (often not encapsulated in tutorial examples)
- small changes, big issues
- hard to grasp concepts from code
- useful defaults mitigate (a bit)



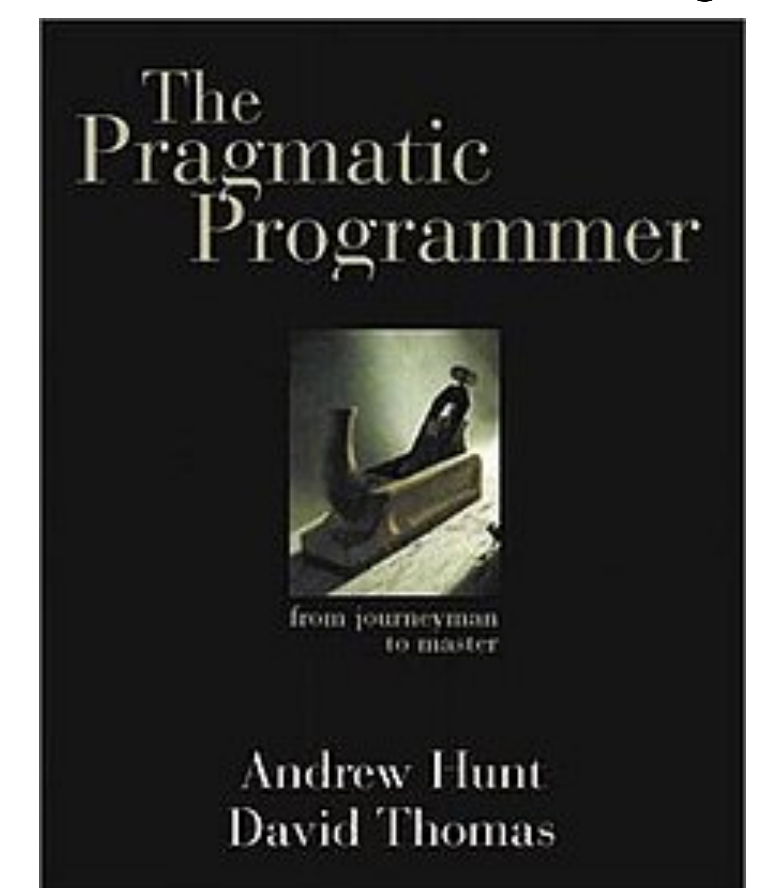

```
cairo_text_extents_t te;
cairo_set_source_rgb (cr, 0.0, 0.0, 0.0);
cairo_select_font_face (cr, "Georgia",
    CAIRO_FONT_SLANT_NORMAL, CAIRO_FONT_WEIGHT_BOLD);
cairo_set_font_size (cr, 1.2);
cairo_text_extents (cr, "a", &te);
cairo_move_to (cr, 0.5 - te.width / 2 - te.x_bearing,
    0.5 - te.height / 2 - te.y_bearing);
cairo_show_text (cr, “a");
```

© Peter Sommerlad

But I need sequencing?

```
last(third(second(first(something))));
```

- Pass the result of what has to be done before as parameter
  - beware of unsequenced evaluation of expression arguments in C++!
- Encapsulate logical sequences
  - ctors of classes, named functions
- Trade-off between genericity, flexibility ⇔ understandability, maintainability
  - stateful API leads to “Programming by coincidence” (The Pragmatic Programmer)



© Peter Sommerlad

When to use sequence matchers

- A stateful, sequence dependent pre-determined API/DOC, like socket()
- The DOC can and will not be refactored
(at least not in the scope of the system you are building)
- The DOC can not be wrapped by a non-sequence-dependent Façade
  - e.g., you are building the wrapper
- You have another good and valid reason that I can not yet conceive
  - e.g., Builder design pattern applied

© Peter Sommerlad

Summary

- If you need to test legacy code - introduce seams and stub DOCs
  - Take all the power C++ and IDEs provide to simplify that as needed
- Only if you have to test the use of a non-changeable stateful API
  - use handwritten or generated mock objects (refactoring friendly!)
- Be aware of the dangers of mocking (frameworks) for new code's design
- Do not mock unwritten code with a mocking framework
  - premature design decisions get glued to the test code in a hard to refactor way

- Remember: KISS applies to test automation as well

© Peter Sommerlad

Cevelop
Your C++ deserves it

Download IDE at:
www.cevelop.com

Sponsors welcome!

Commercial licensing possible!

Questions?

peter.sommerlad@hsr.ch
@PeterSommerlad

© Peter Sommerlad

By the way, thanks for the great piece of software! This is by far the best free IDE for C/C++ so far after trying basically all the free C/C++ IDE. motowizlee on github 27.04.2017