On reflecting on runtime or, *"Program know thyself"*

ACCU 2015 Bristol

Dominic Robinson

dominic_robinson@sn.scee.net

Part I *of V*

Navel gazing

Part II of V

Navel gazing

Existential C++

Part III of V

Navel gazing
Existential C++
Genesis of Intent

Part IV of V

Navel gazing
Existential C++
Genesis of Intent
Archaeology

Part V of V

Navel gazing
Existential C++
Genesis of Intent
Archaeology
Agent Provocateur

Part Zero

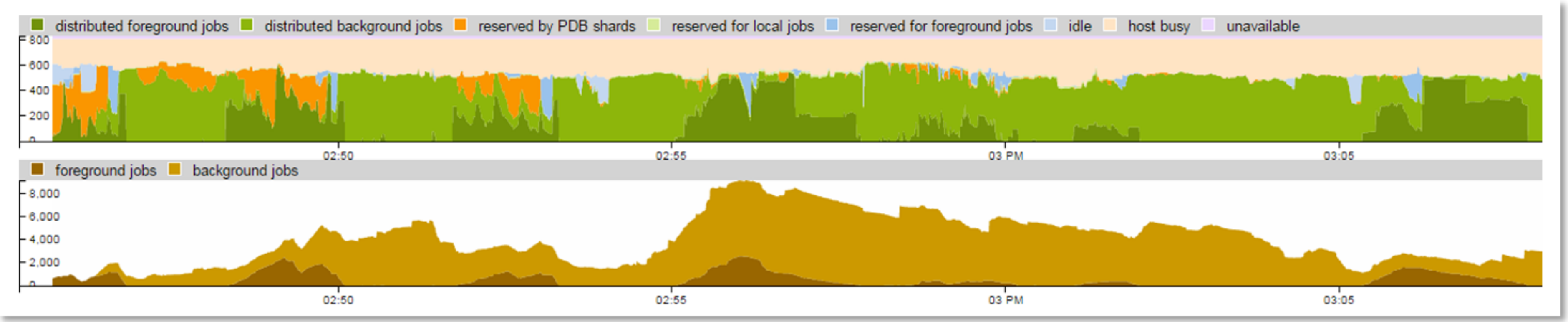
Context

What do I do?

- A distributed build accelerator
- Written in C++ in the style of Erlang
- Runs on tens to hundreds of machines
- Big enough to fail in *interesting* ways

What does it do?

- It distributes compilation and data processing
- Here it is keeping 600 cores busy on up to 8,000 simultaneous jobs for 30 minutes



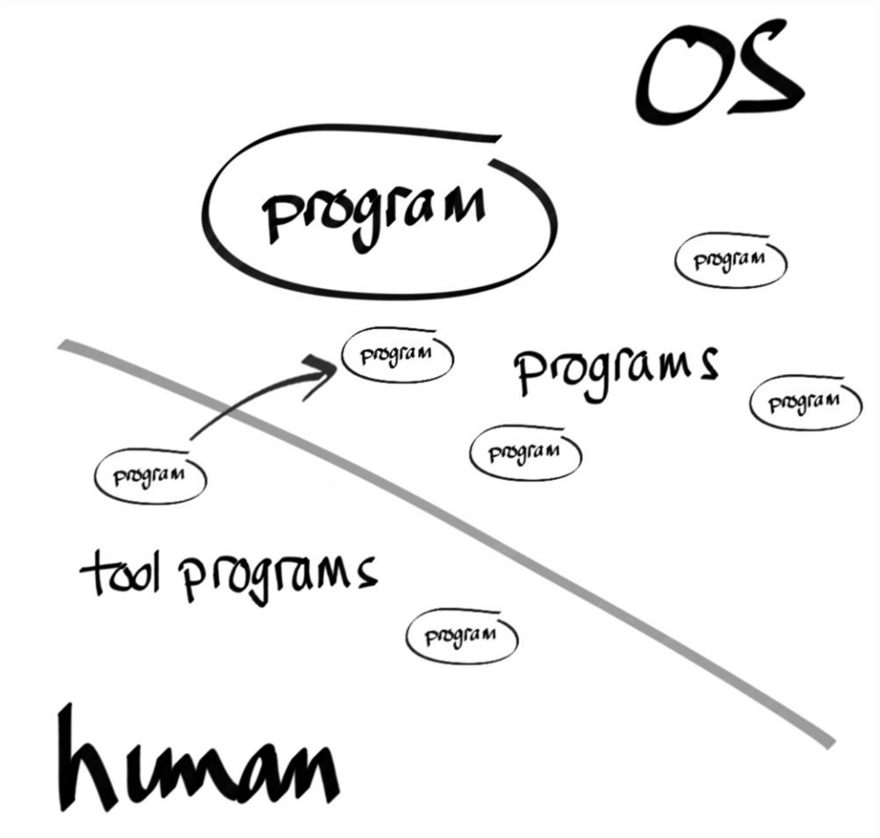
Part I

Navel gazing

- What is the "self"?
- What is "runtime"?
- What is "reflection"?

Navel gazing

What is
the self?



What is
the self?

What is runtime?

The Seven Ages of Code

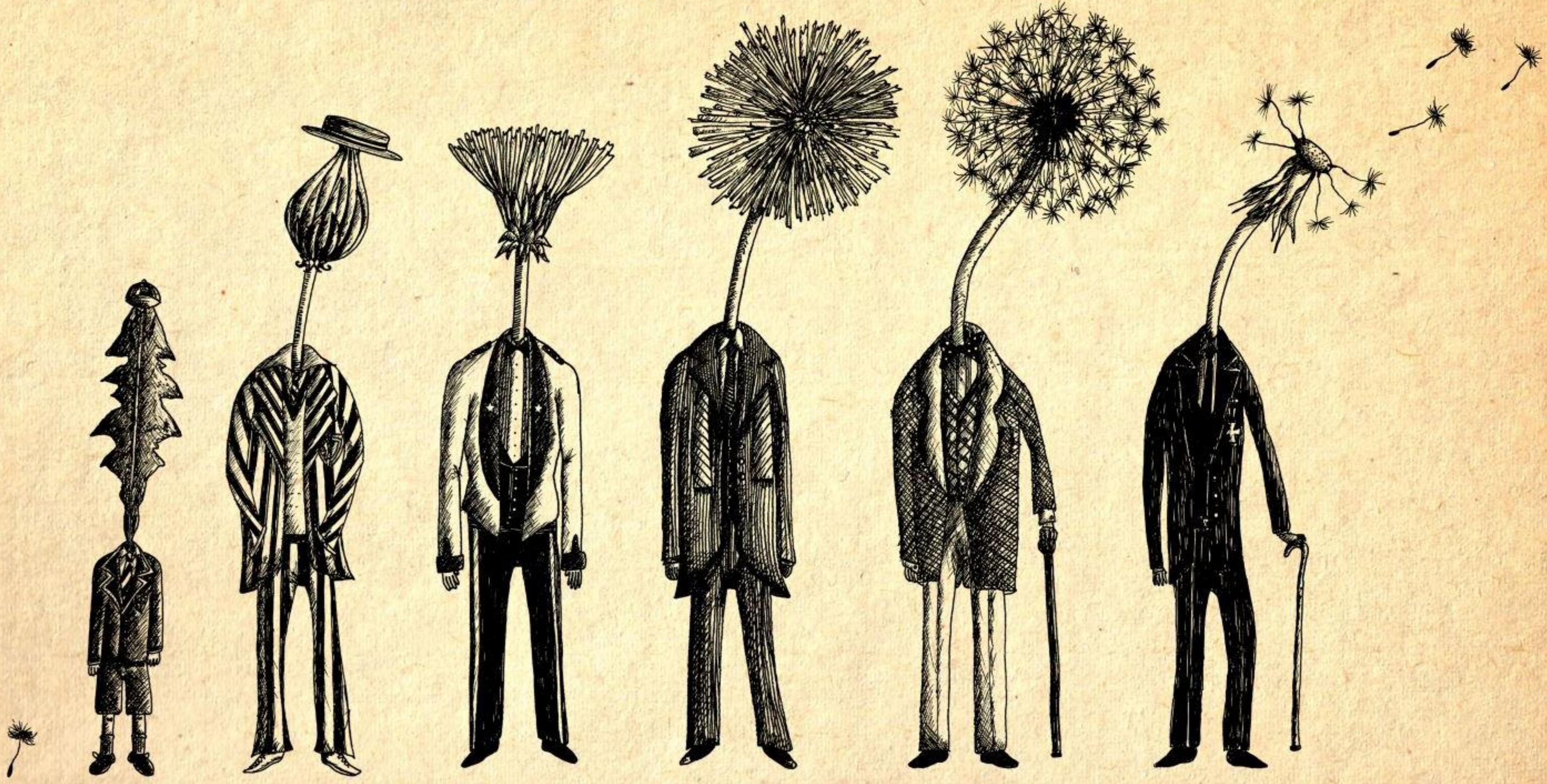
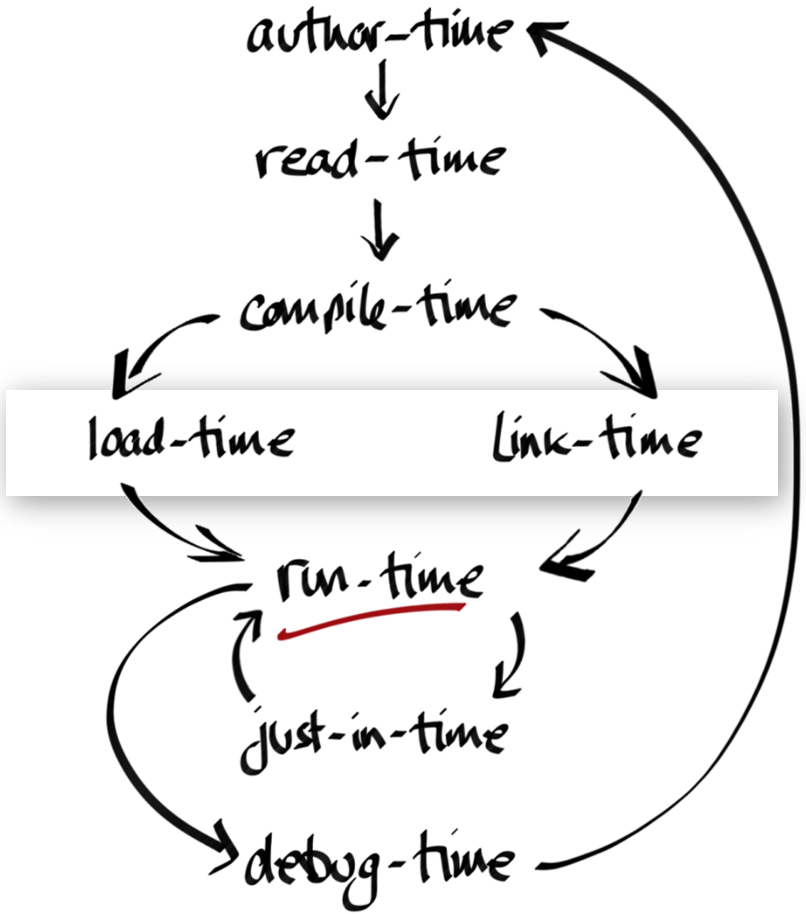
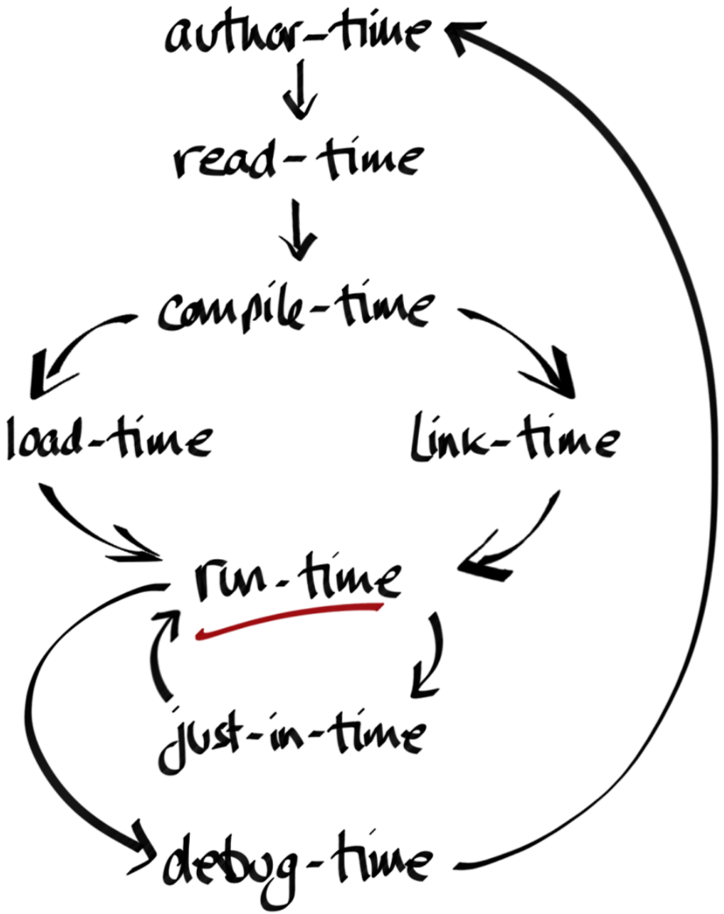


fig. 3: The Seven Ages of Mandelion

By kind permission of Jon Turner, http://www.thisisjonturner.com

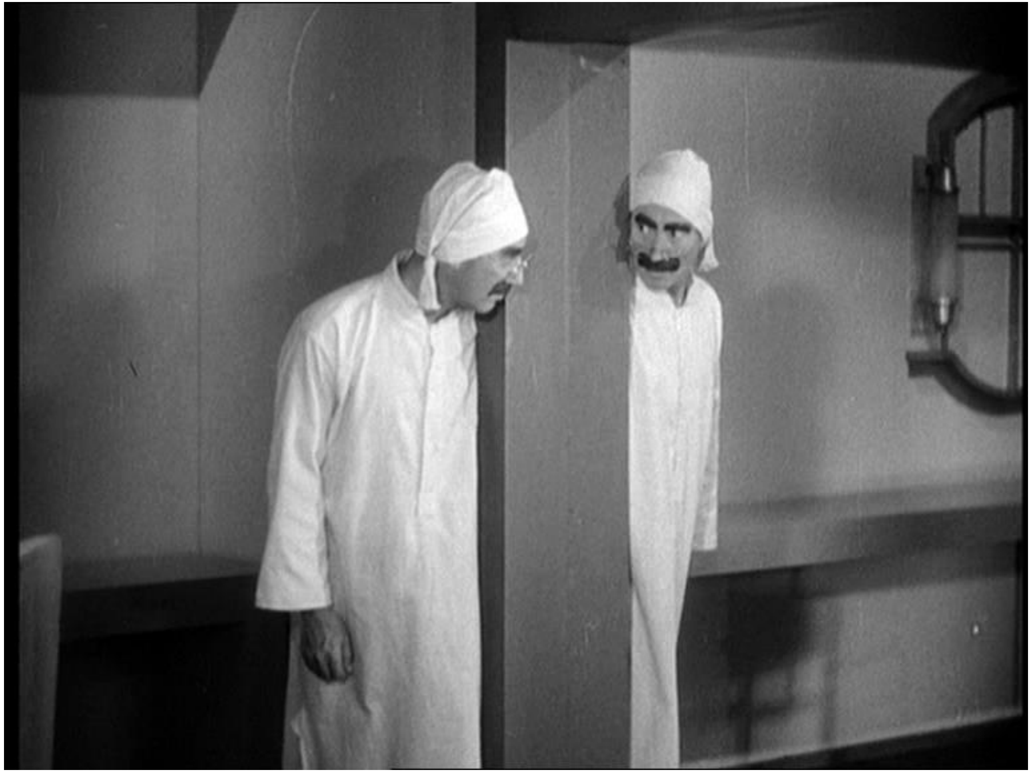


The Seven Ages of Code



The Eight Ages of Code

What is reflection?



What is reflection?

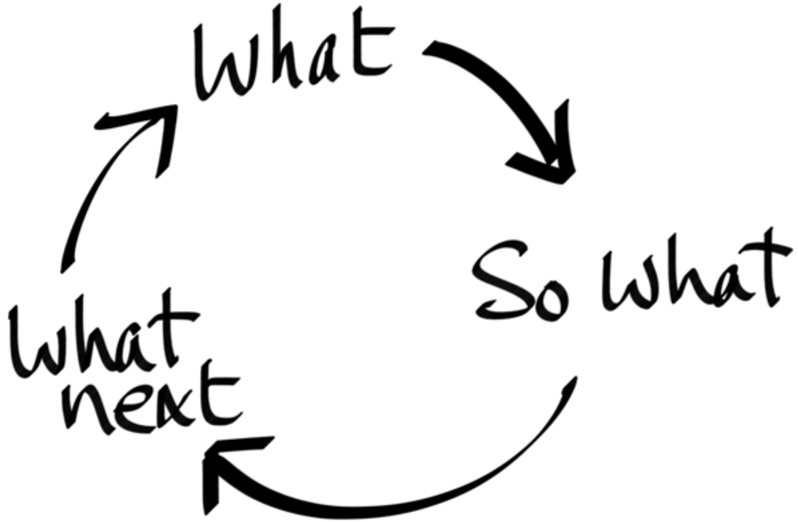
What is reflection?

- Reification :
  - making the implicit visible
  - to convert into or regard as a concrete thing

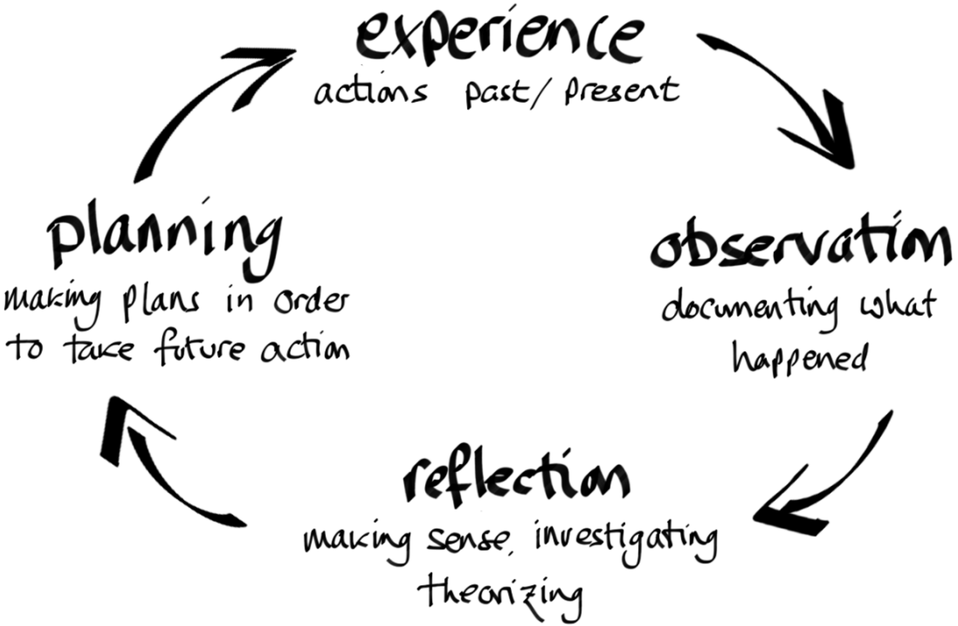
What is reflection?

- Is that it?

Reflective Practice



Reflective
Practice



Reflective Practice

Do programs
practice
reflective
practice?

- **Through a glass darkly :** Shedding light on reflective practice and autonomous learning

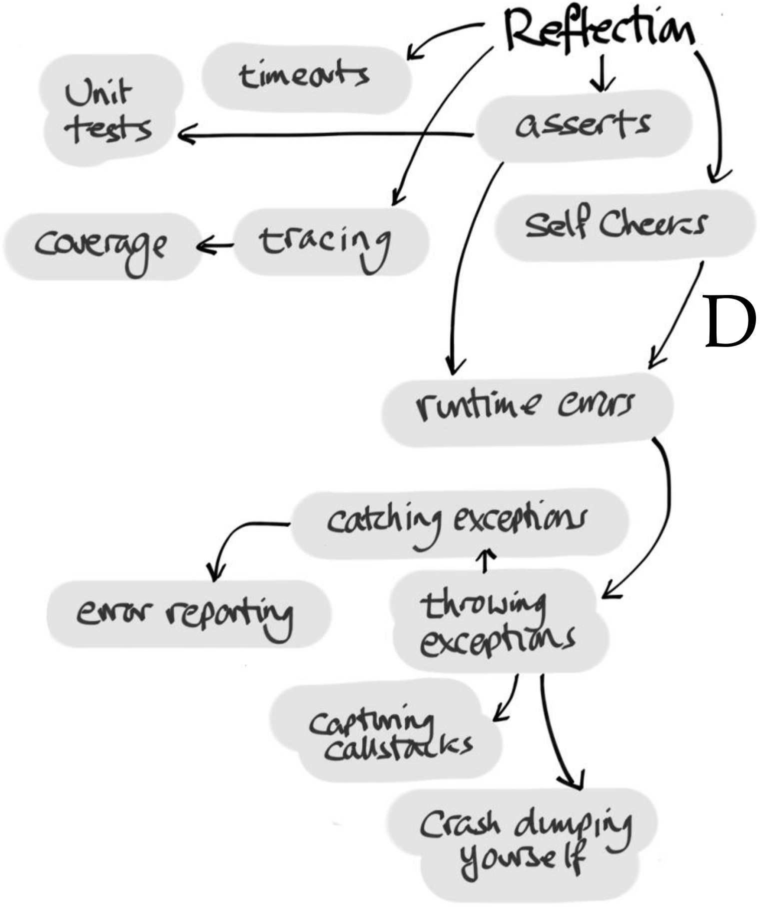
Reflective Practice

Reflective Practice

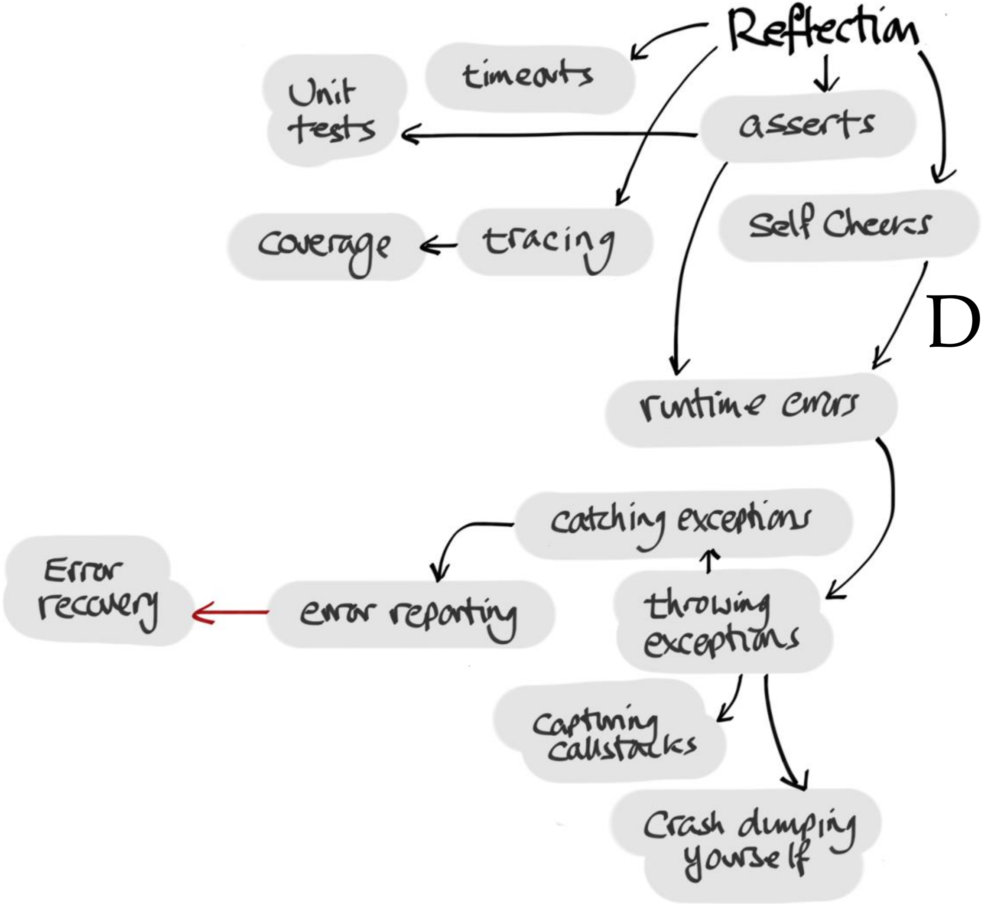
- **Through a glass darkly :** Shedding light on reflective practice and autonomous learning

- *"Reflection may not be enjoyable but it is recorded as a non-threatening process, which can include a balance of positive and negative experiences and has a significant value for students especially in learning from their mistakes."*

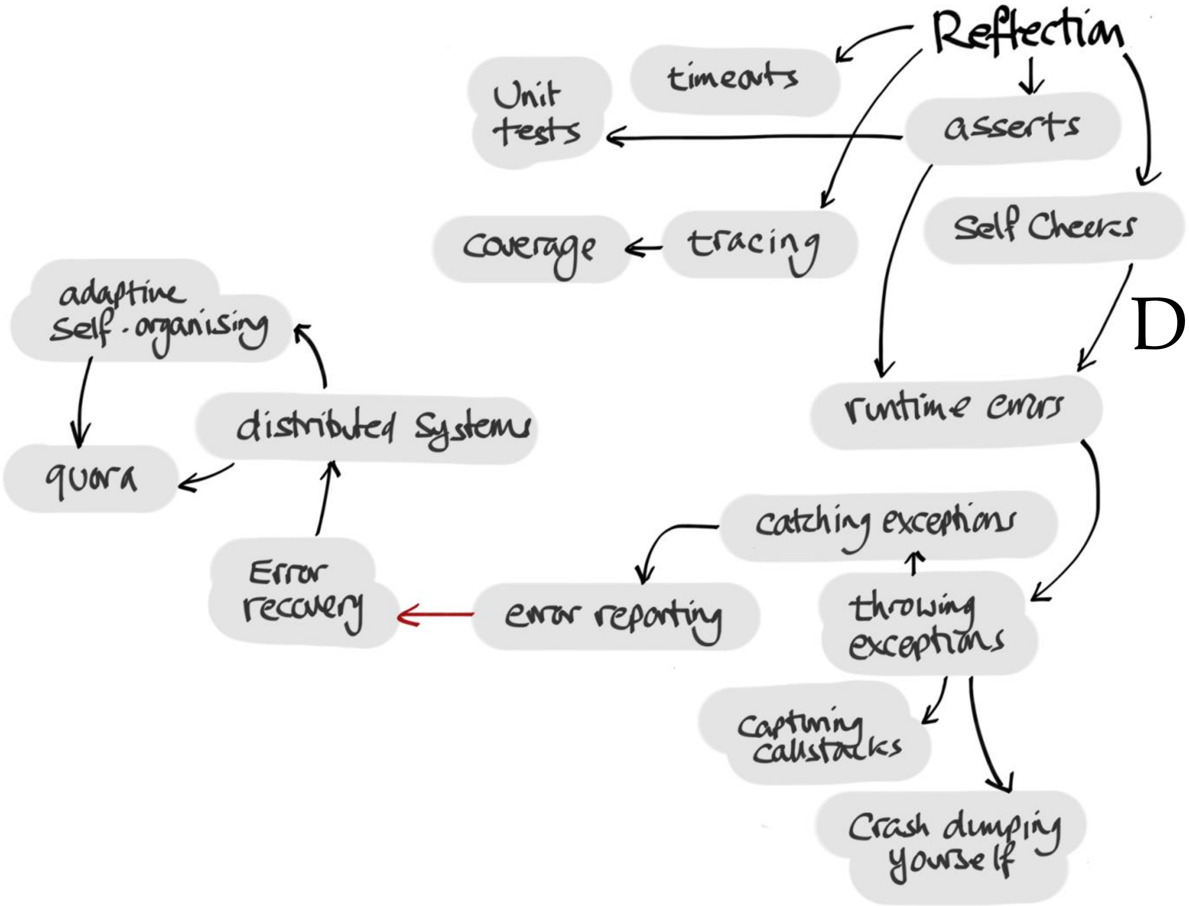
- Susan M Taylor and Mary A Dyer, University of Huddersfield, 2010 (unpublished)
- http://eprints.hud.ac.uk/8408



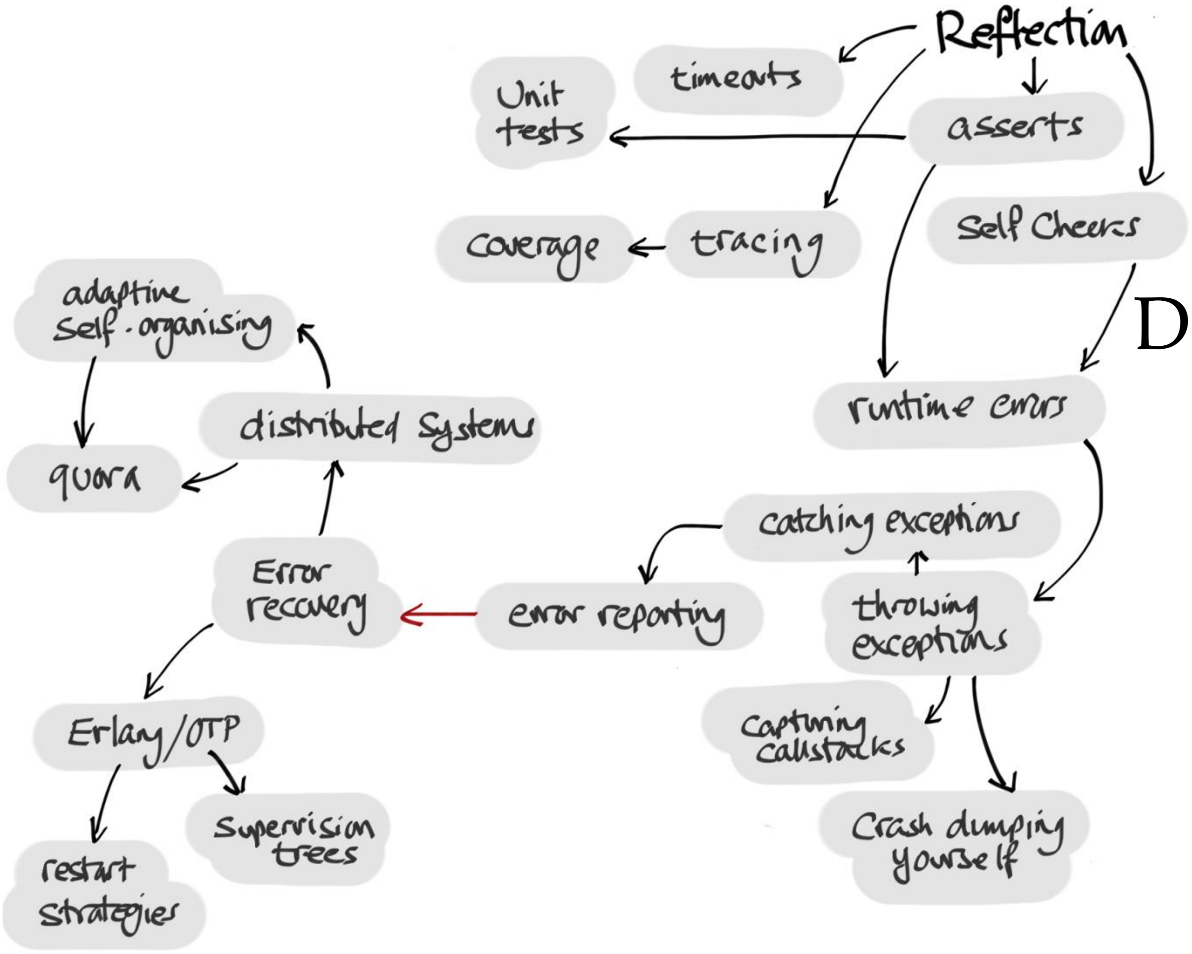
Do programs practice reflective practice?



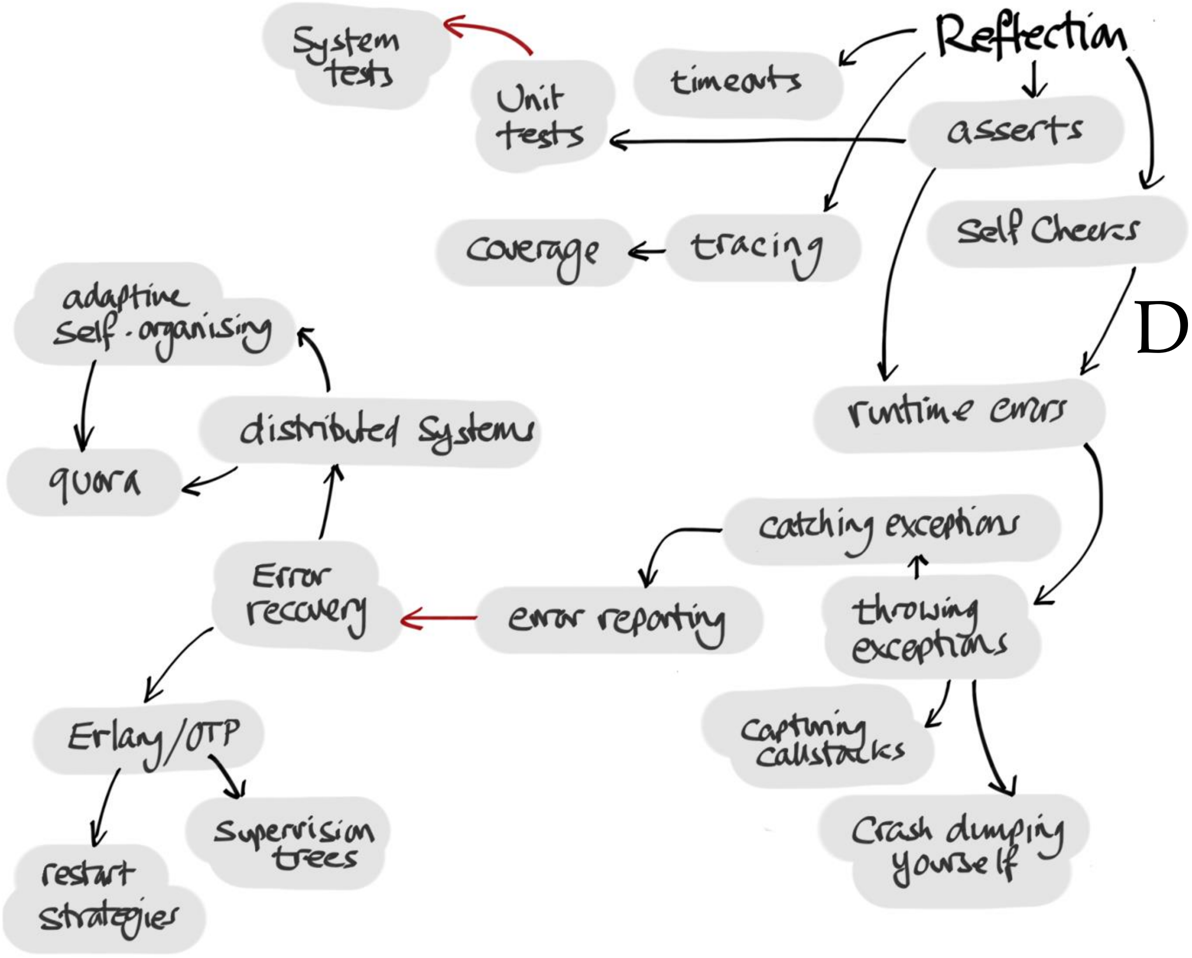
Do programs practice reflective practice?



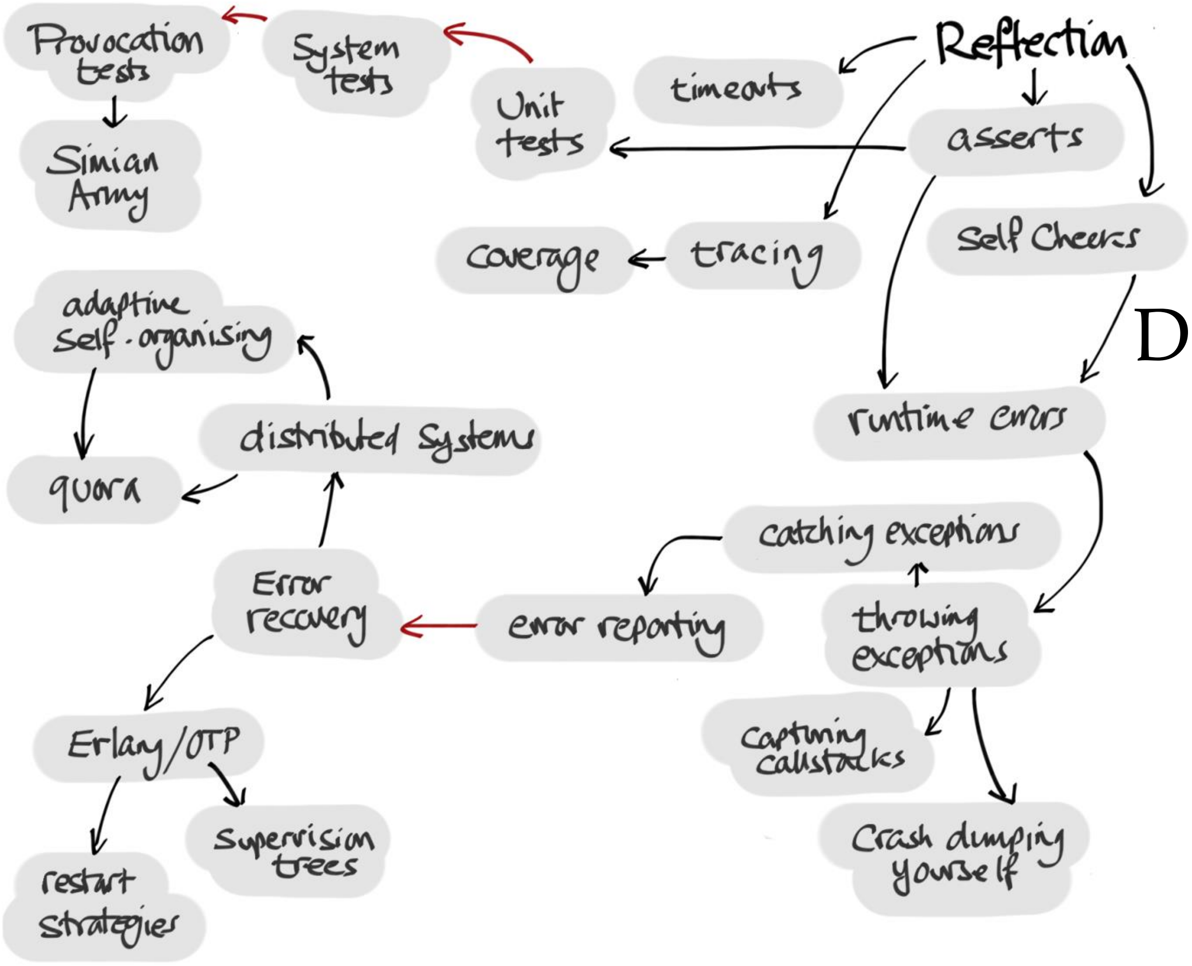
Do programs practice reflective practice?



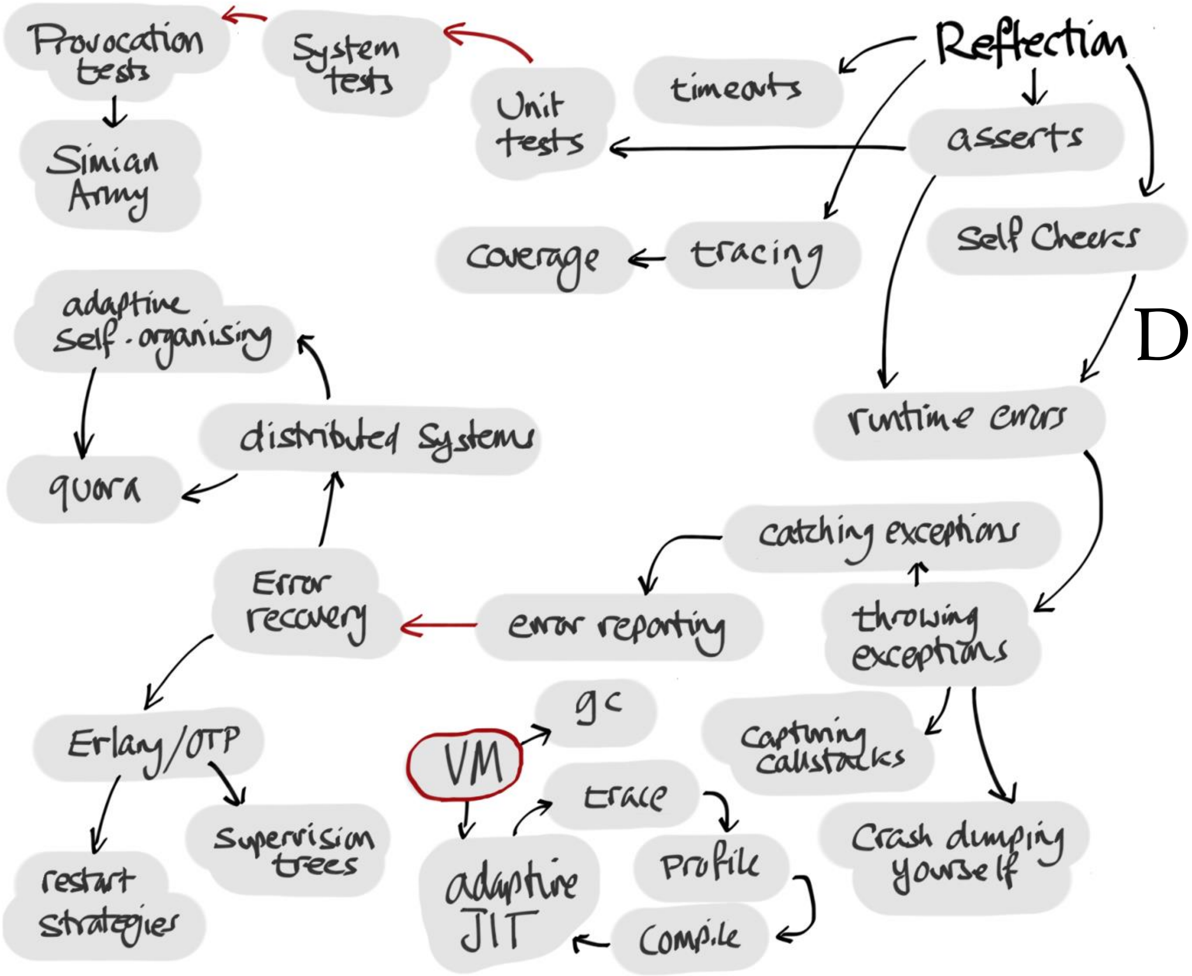
Do programs practice reflective practice?



Do programs practice reflective practice?



Do programs practice reflective practice?



Do programs practice reflective practice?

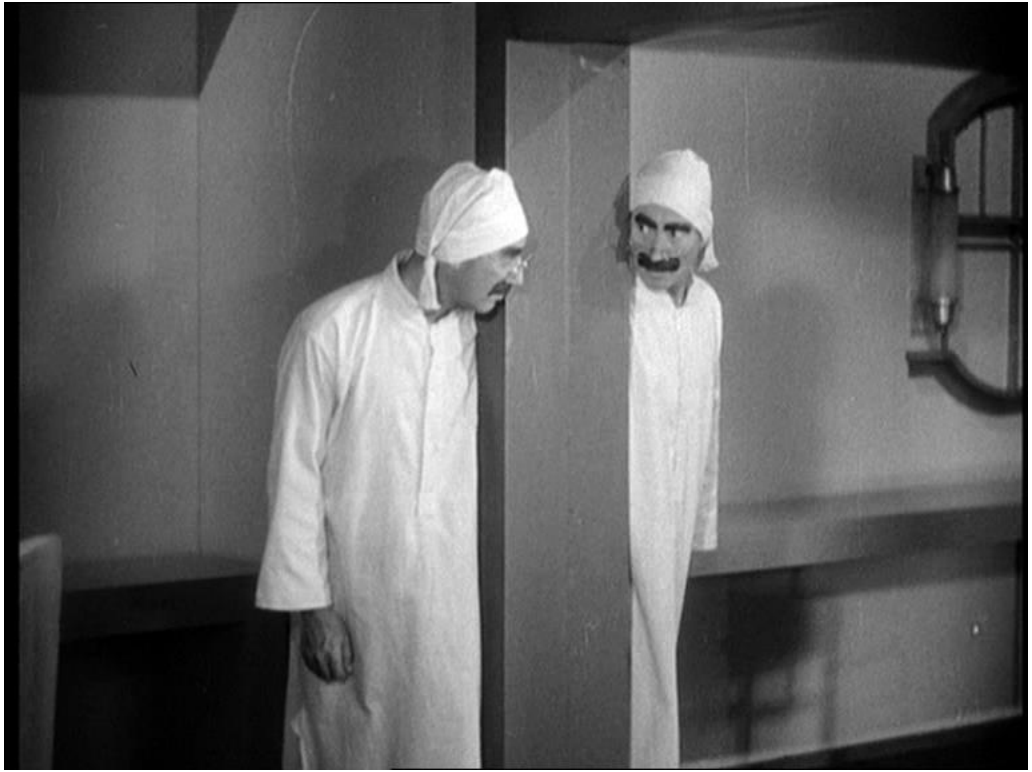
So, what about
C++?

Part II

Existential C++

- A C++ program’s experience of execution

Existential C++



What is reflected?

Shadow puppets

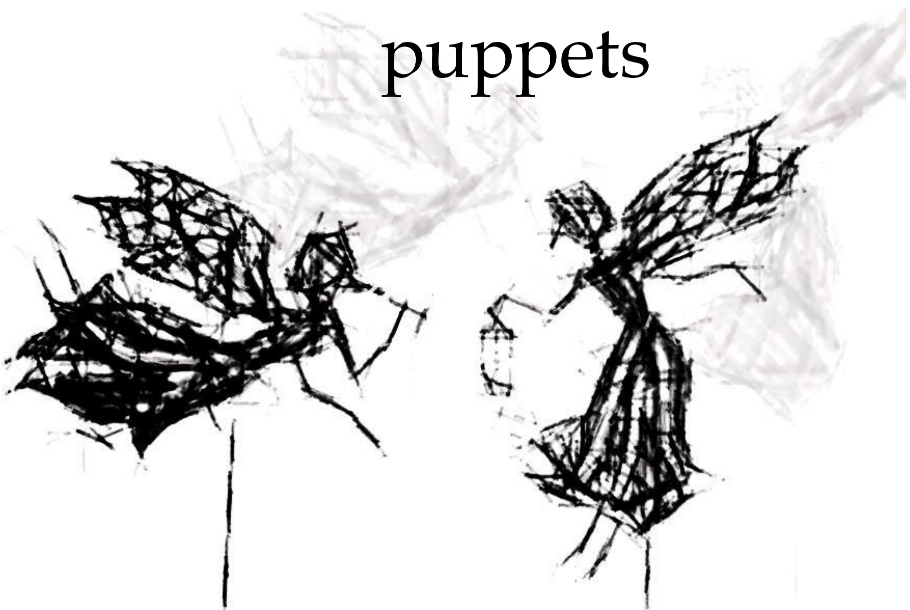
- The semantics of C++ are projected onto the hardware execution model.

- They are implemented behind the screen by representation artefacts.

- *http://papertales.deviantart.com/art/Two-Angels-Shadow-Puppets-174367145*

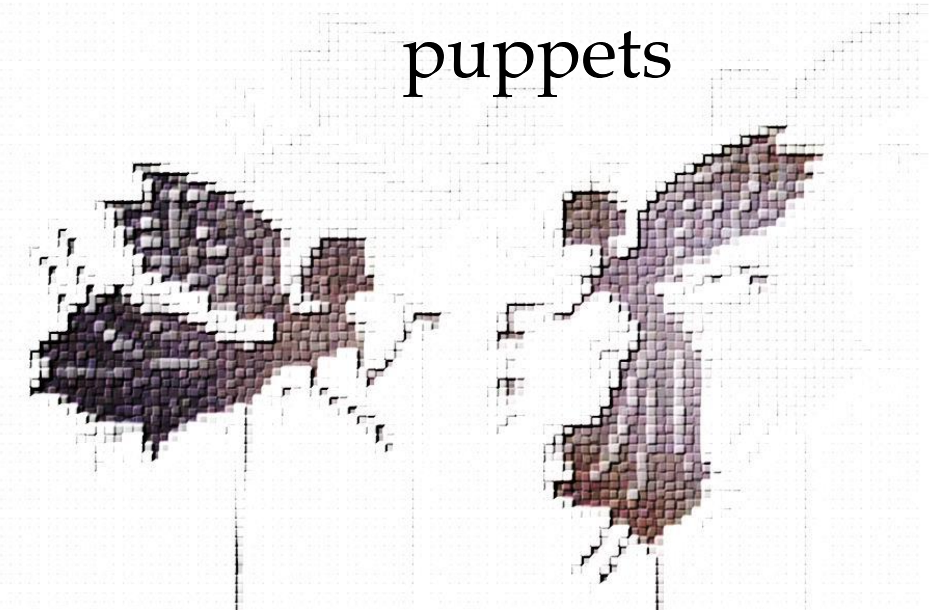
- The semantics of C++ are projected onto the hardware execution model.
- They are implemented behind the screen by representation artefacts.
- Intel doesn't want you to know that in most cases these are wood and string.

Shadow puppets



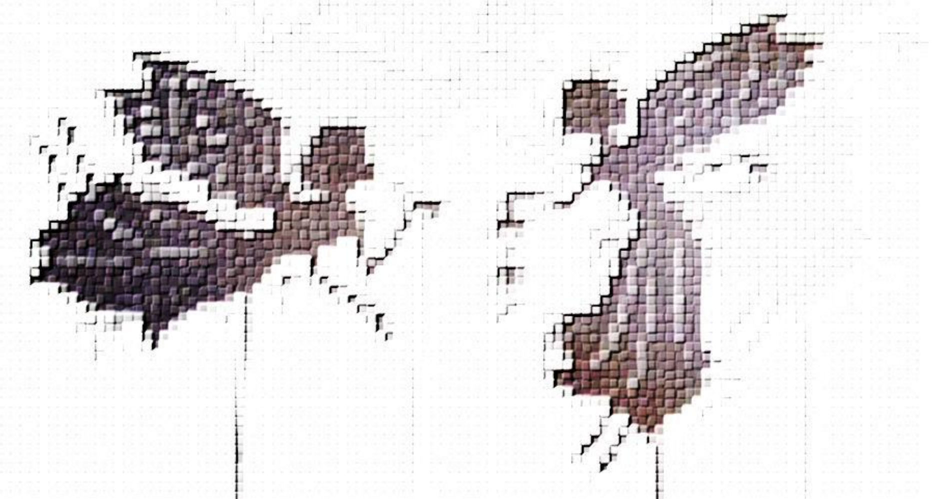
- What can we see?
  - Inspect values that are in scope
  - Inspect memory, perhaps interpret it by heap walking
    - Memory leaks
    - Memory corruption
  - Inspect objects using a MOP
  - Inspect objects using a DWARF

Shadow puppets



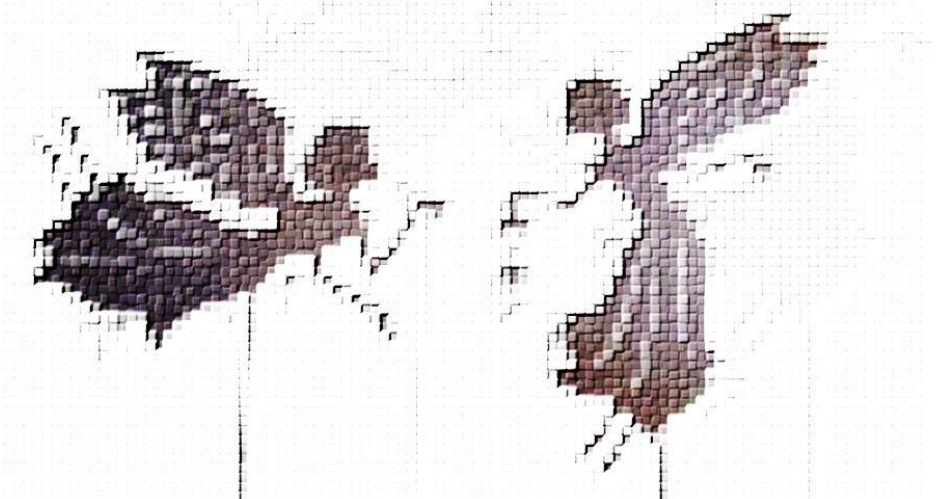
Shadow puppets

- What can we measure?
  - Resource usage
  - Work done against time
    - Timeouts
    - Profiling
    - QOS guarantees



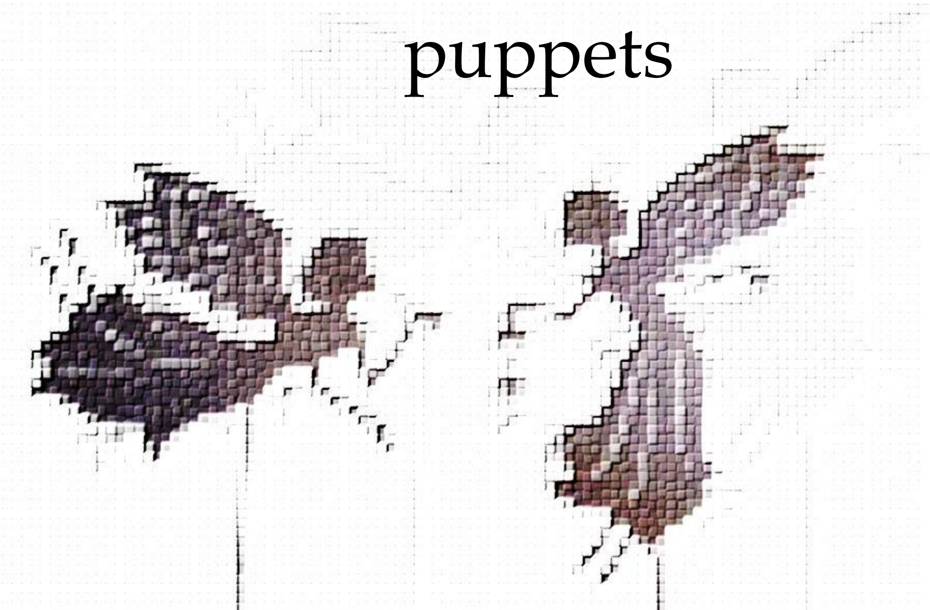
- What can we capture?
  - History
    - Execution history using logs and traces (`printf`)
    - Call stacks (requiring debug data to decipher)
    - Exceptions
    - Core dumps to snapshot state

Shadow puppets



- What is least well represented, or taken for granted?

Shadow puppets



Execution flow

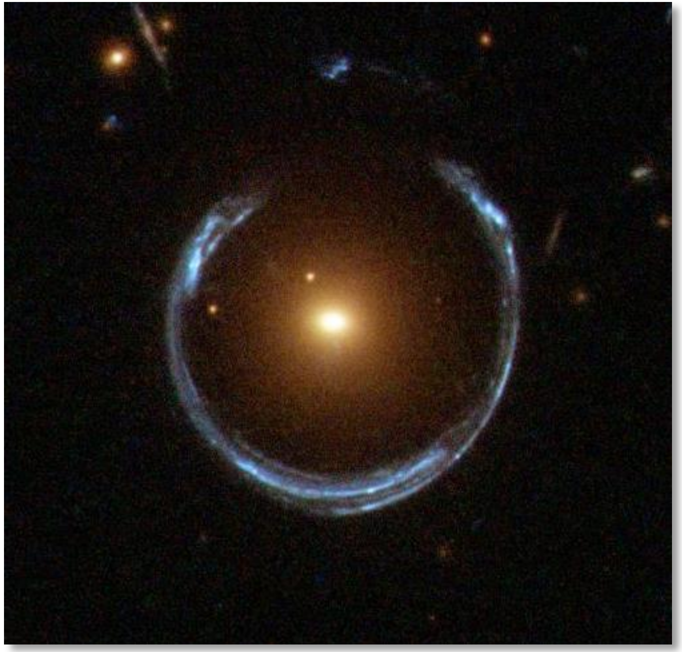
What is
The Standard
Model?

Execution flow

- Stack based model
  - Lexical scopes
  - Call and return
  - Exceptions and unwinding

Concurrent Execution flow

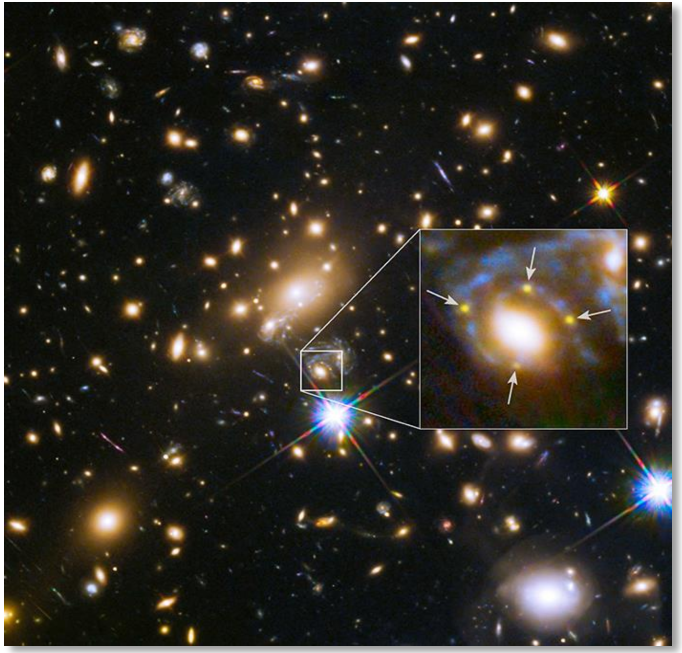
- As parallelism and concurrency become more prevalent, the execution of work related to a domain thing may no longer follow the familiar call stack model.
- Work queues, thread pools, co-routines, message passing, actors, and distributed systems all cause work fragments to be scattered, becoming disconnected.



A metaphor…

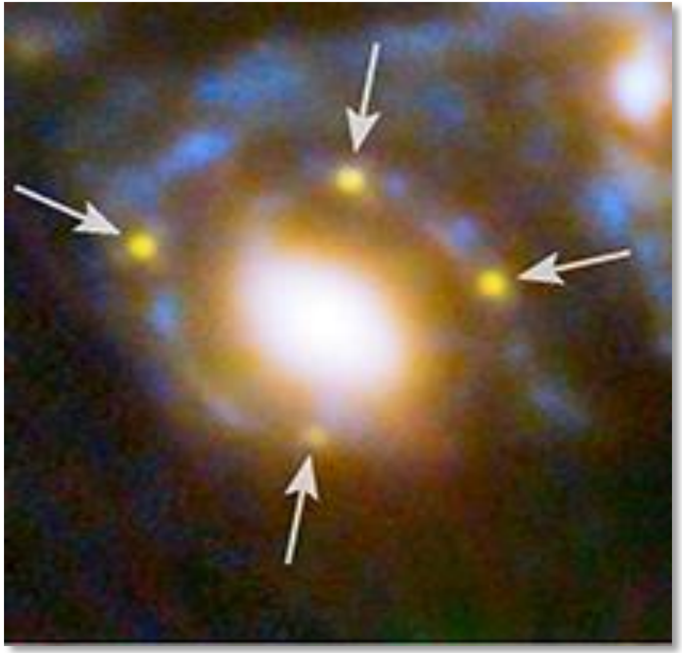
Einstein's Gravity Lens

- http://upload.wikimedia.org/wikipedia/commons/1/11/A_Horseshoe_Einstein_Ring_from_Hubble.JPG



Einstein Cross

- http://physicsworld.com/cws/article/news/2015/mar/05/gravitational-lensing-creates-einsteins-cross-of-distant-supernova



- http://physicsworld.com/cws/article/news/2015/mar/05/gravitational-lensing-creates-einsteins-cross-of-distant-supernova

Einstein Cross

- Conventional control flow is becoming less well correlated with domain work.

Execution flow

The Fabric of
Space and Time
is under threat!

`call/cc`

The Fabric of
Space and Time
is under threat!

A glimmer of hope

- C++ 11’s transportable exceptions are a reaction to new execution flow models.
- Exceptions are becoming first class objects.
- Exception flow can be manipulated.
- Errors can be captured and propagated across between execution fragments to maintain their association with work items.
- Applications have to work at this.

- More generally…

Causality

Causality

- *the relationship between something that happens or exists and the thing that causes it*

- *cause* and *effect*

Causality

- If execution flow is what enacts *cause* and *effect,* how is this made manifest?

Effect

- Programs do work to compute values.
- Doing **work** gives rise to *values* or *exceptions*.

- *effect* = *values* **or** *exceptions*

Effect

- *Systematic Error Handling in C++ 11*

- Andrei Alexandrescu describes the use of **Expect<T>** to unify the handling of results or the exceptions incurred whilst attempting to compute them.

- Expect<T> encodes a *value* or an *exception*.

- What Expect<T> encodes is *effect*.

- *C++ and Beyond 2012 http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Andrei-Alexandrescu-Systematic-Error-Handling-in-C*

❖ `Expect` is *effect* made manifest:

```
template <class T> class Expect {
  union {
    T ham;
    std::exception_ptr spam;
  };
  bool gotHam;
  ...
}
```

Expect

Promises

- C++ 11 Promises go a step further by promising to represent the results (values or exceptions) of computation that may not yet have completed.
- *future effect*

Promises, promises

- The ability to represent the future results of work is a step towards *execution flow meta-programming*.
- But, C++11’s promises are missing the composability that would enable programs to construct, observe and manipulate their execution own flow.

- *See, for example the Promises/A+ spec from the javascript world:*
  https://promisesaplus.com
  and:
  http://bartoszmilewski.com/2009/03/03/broken-promises-c0x-futures/

- What then of *cause*?

Causality

- It must be manifest in the **work**.

- Programs perform the **work** by calling functions that return values or throw exceptions.

- But functions are complex implementation artifacts. They are too unconstrained to be readily reflected upon and understood.

Cause

- Let's look for *inspiration*…

Cause

Insight

- Andrei Alexandrescu identified a key insight:

  "Error codes are limited, exceptions are arbitrarily rich.

  Make exceptions be the error codes."

- *C++ and Beyond 2012 http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Andrei-Alexandrescu-Systematic-Error-Handling-in-C, slide 12.*

Insight

- … but I think there was something on the previous slide:

  “Exceptions are associated only with root reasons, not goals.

  '*I/O error*' *doesn’t describe* '*saving weight file*'.*”*

- *C++ and Beyond 2012 http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Andrei-Alexandrescu-Systematic-Error-Handling-in-C, slide 11.*

- … but I think there was something on the previous slide:

  "Exceptions are associated only with root reasons, not ***goals****.*

  '*I/O error*' *doesn't describe* '*saving weight file*'*."*

Insight

- *C++ and Beyond 2012 http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Andrei-Alexandrescu-Systematic-Error-Handling-in-C, slide 11.*

Exceptions re-examined

Exceptions re-examined

- Exception handling is also execution flow control, albeit backwards.
- It has fewer degrees of freedom.
- Scary documents extol narrow best practice: *don't, no really don't,* ***or else****...*
- http://www.boost.org/community/error_handling.html

- In other words:

  "When an exception is thrown I shall smite thee back to the dark ages."

Taking exception

- In other words:

  "When an exception is thrown I shall smite thee back to the dark ages."

  "Thou shalt not use std::string."

Taking exception

- In other words:

  “When an exception is thrown I shall smite thee back to the dark ages.”

  “Thou shalt not use std::string.”

  “Thou shalt pre-allocate buffers for text and use strcpy.”

Taking exception

Taking exception

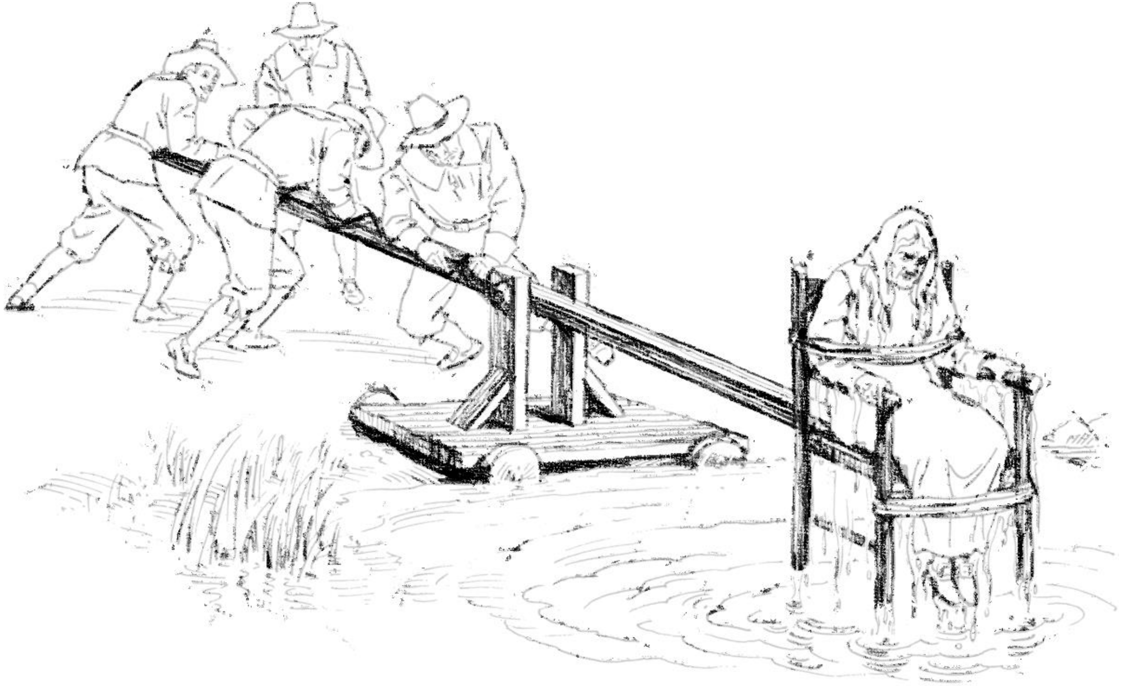
- In other words:

  "When an exception is thrown I shall smite thee back to the dark ages."

  "Thou shalt not use std::string."

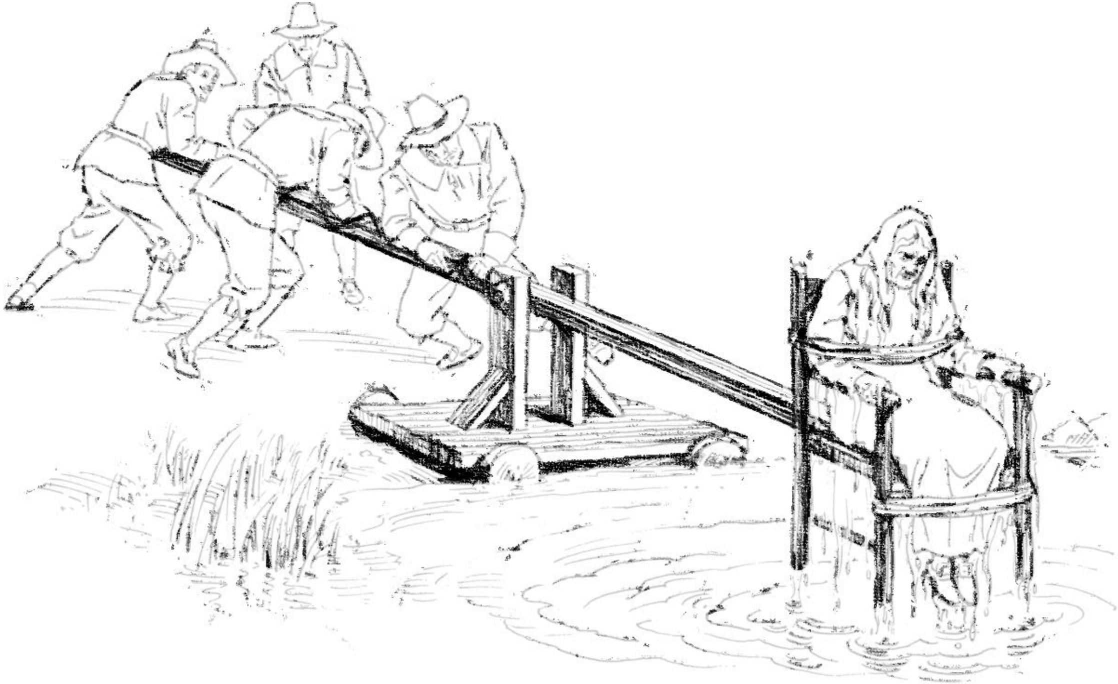
  "Thou shalt pre-allocate buffers for text and use strcpy."

  "Thou shalt not be tempted by opportunities for exotic flow control."



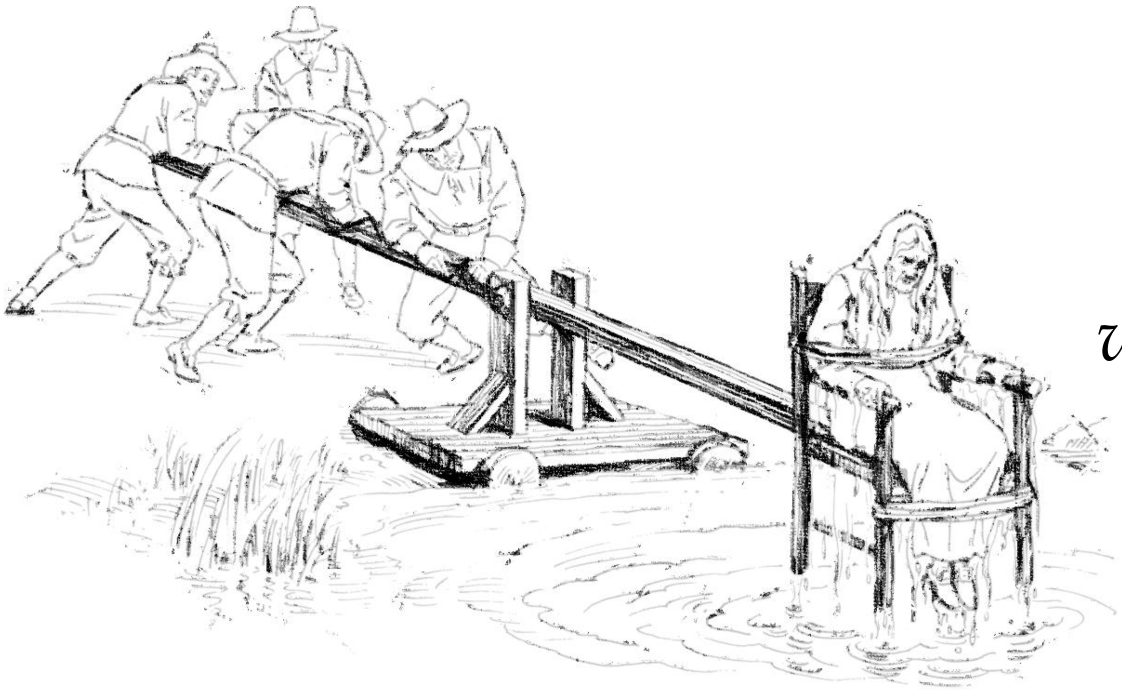
Or else…

- *Adapted from http://en.wikipedia.org/wiki/Cucking_stool#/media/File:Ducking-Stool_1_(PSF).png*



relax…

- *Adapted from http://en.wikipedia.org/wiki/Cucking_stool#/media/File:Ducking-Stool_1_(PSF).png*



because
we're made of
sterner stuff

Exceptions re-examined

- Exceptions are *out of band*, invisible to intervening code.
- We talk about code being *transparent to exceptions*.

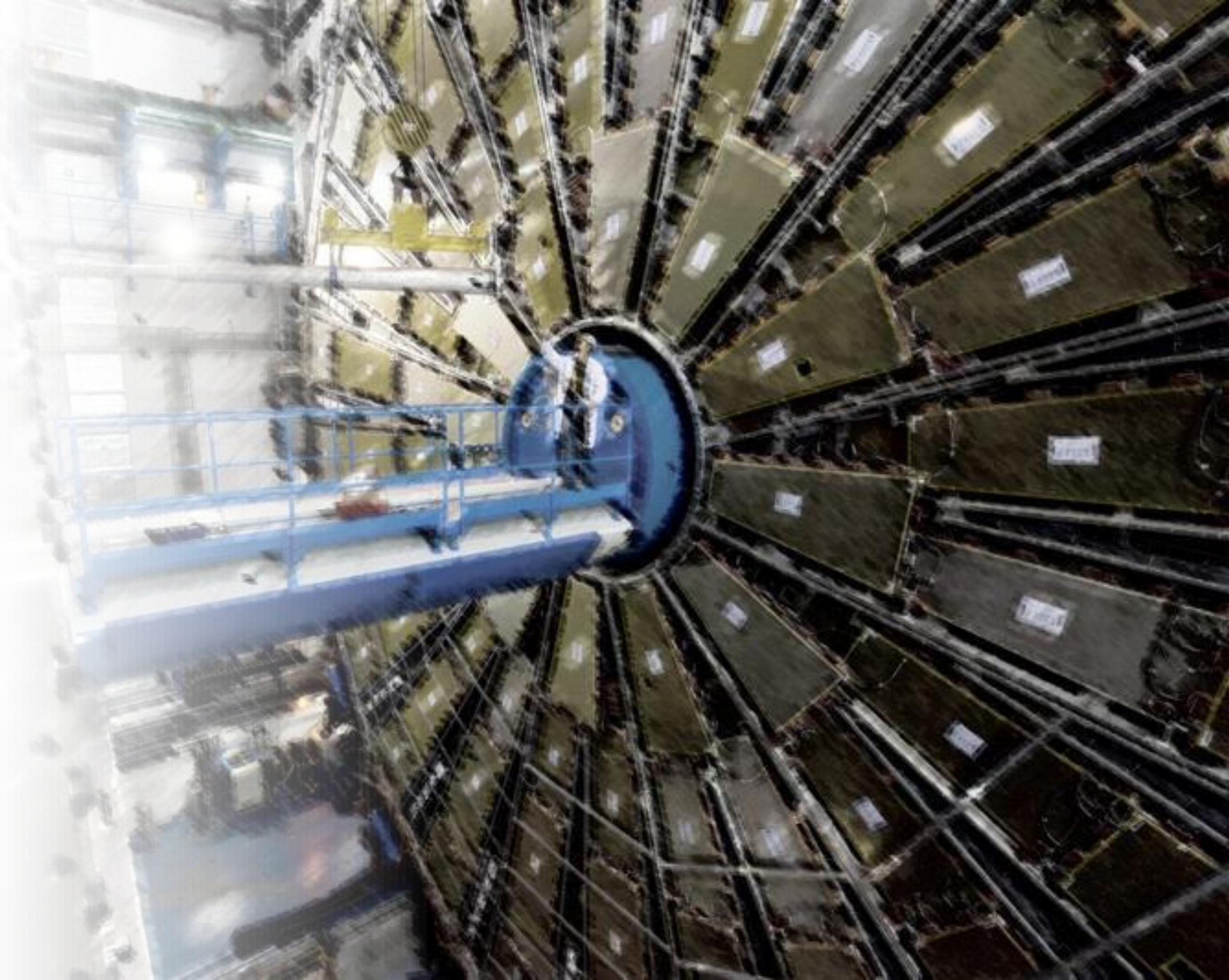
Exceptions re-examined

- Yet the resulting execution flow **can** be observed by suitably constructed detector.

Exceptions re-examined

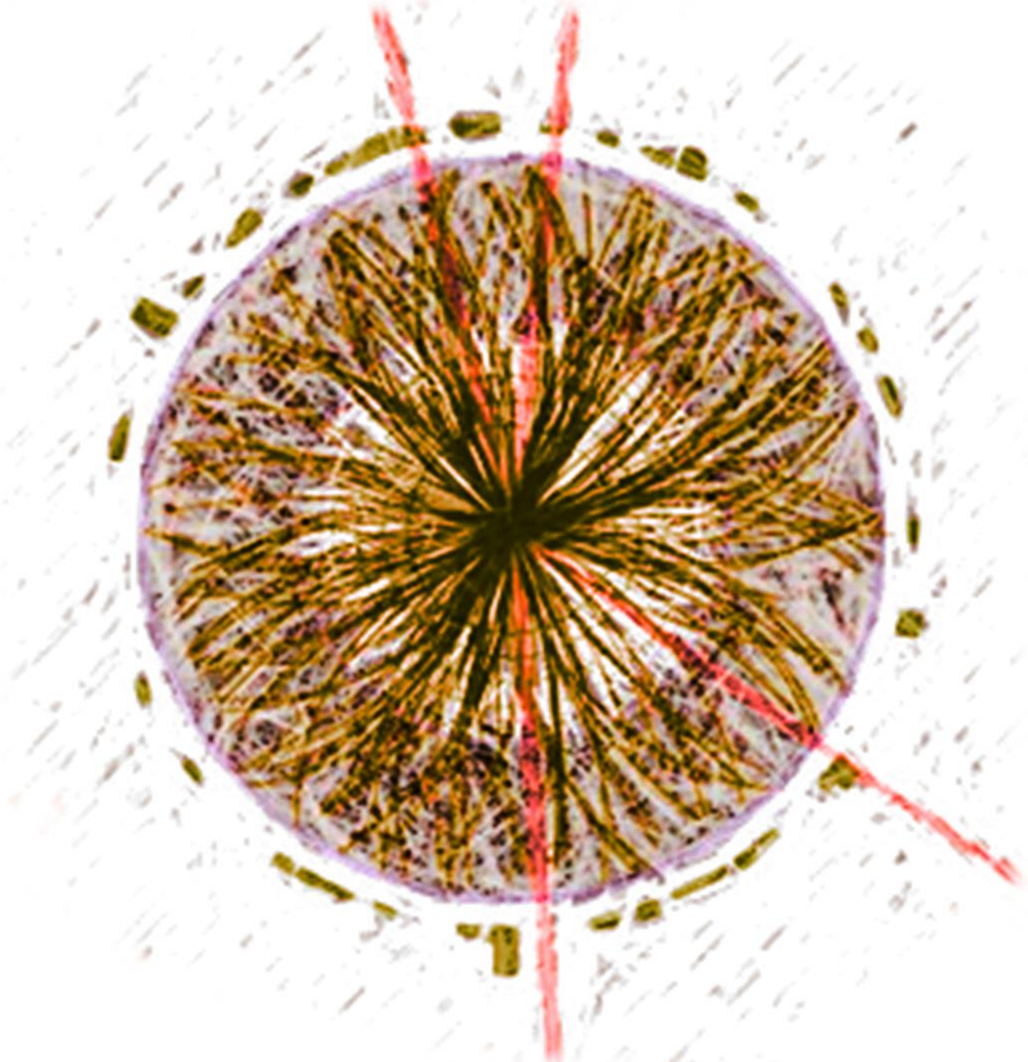
- Luckily Axel Naumann from CERN was here yesterday…

❖ And lent me some spare parts



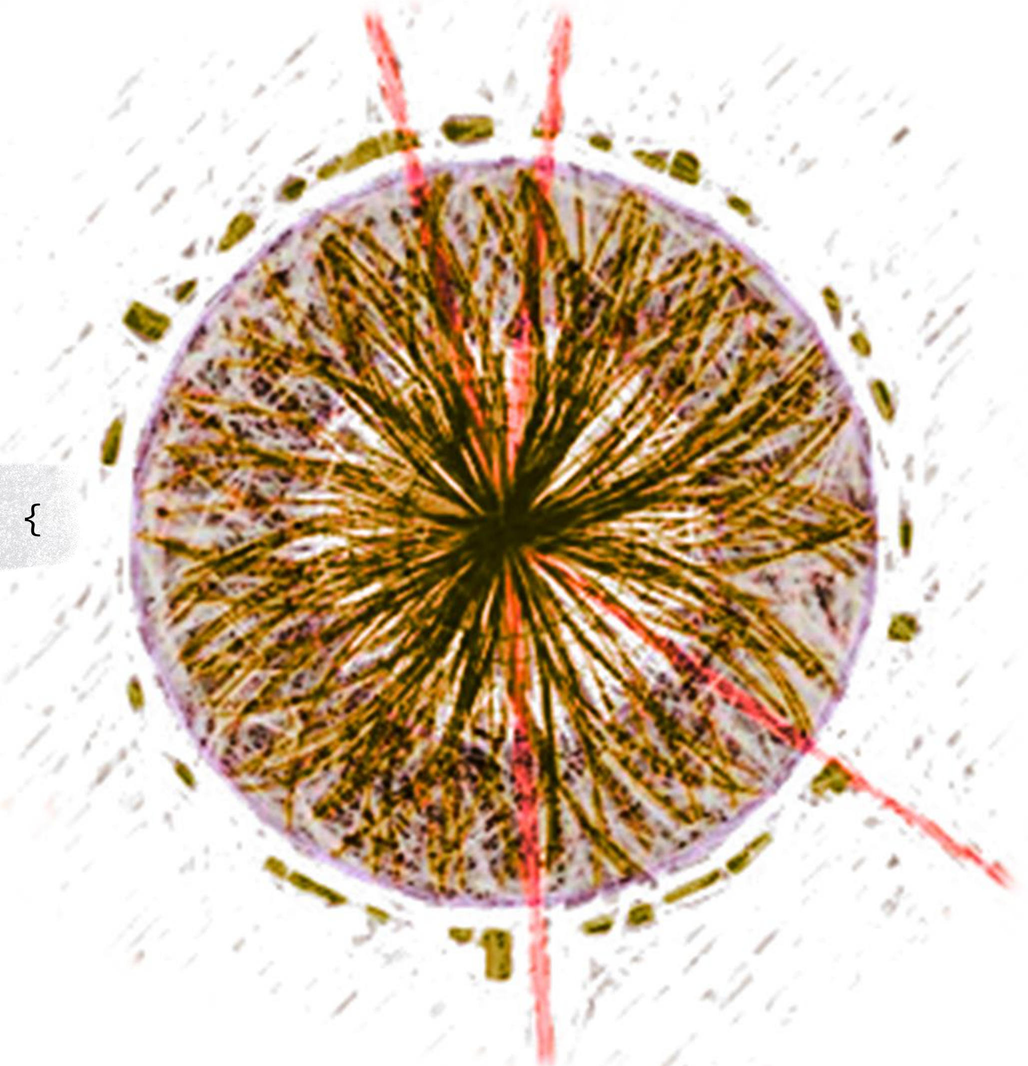
❖ *Adapted from photo: © CERN*

- Exception detector

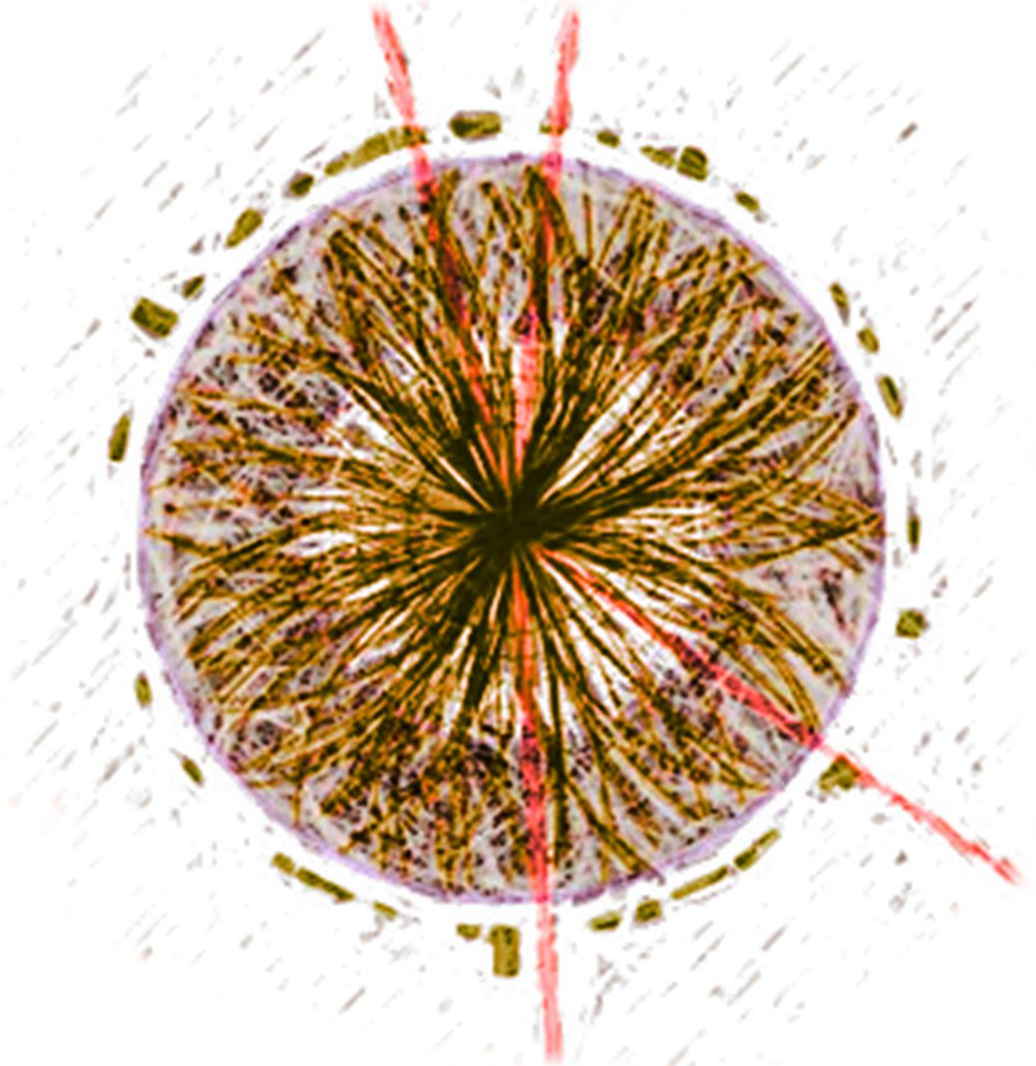


- *Adapted from ATLAS Experiment © 2012 CERN*

```
detector() {
  entering a scope
};

~detector() {
  leaving a scope
  if (std::uncaught_exception()) {
    exceptionally
  } else {
    normally
  }
};
```

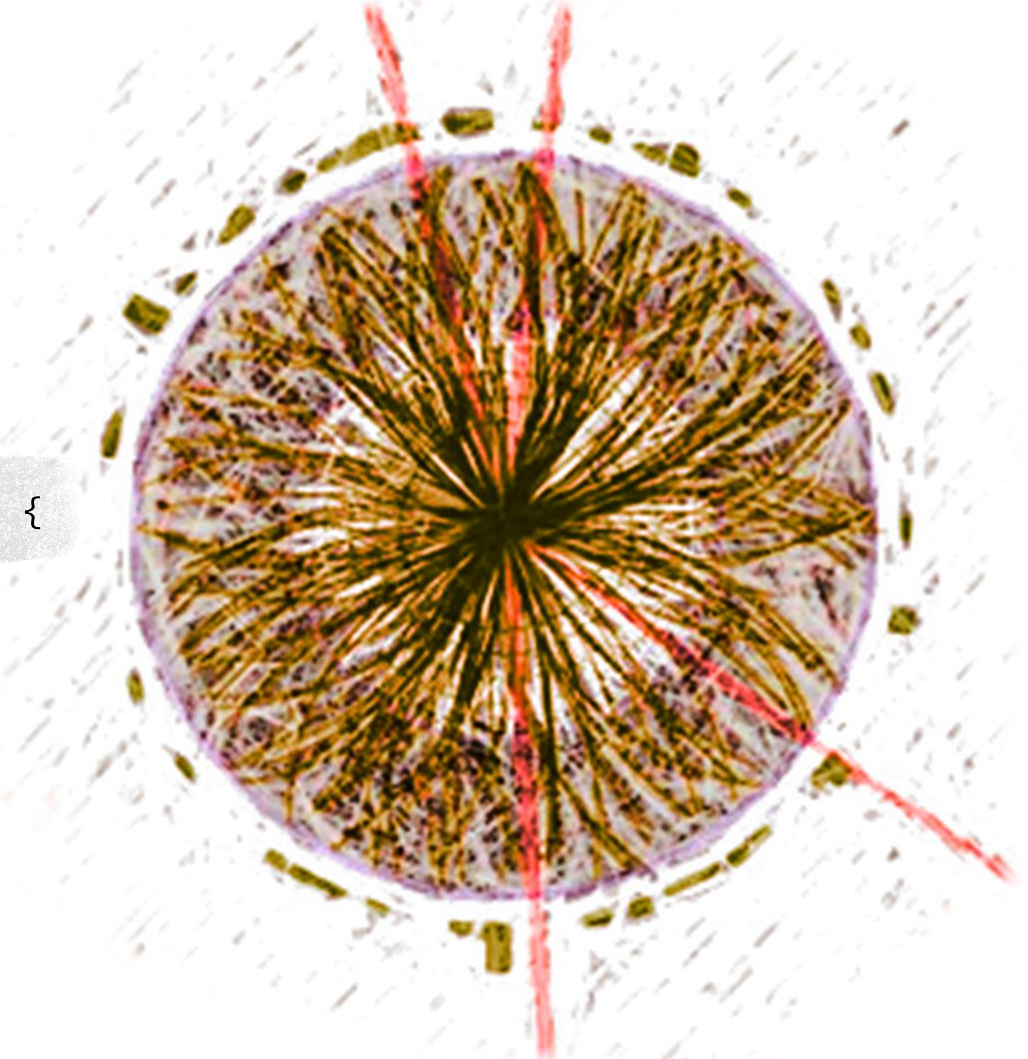


- Is this detector safe?



```
detector() {
  entering a scope
};

~detector() {
  leaving a scope
  if (std::uncaught_exception()) {
    exceptionally
  } else {
    normally
  }
};
```

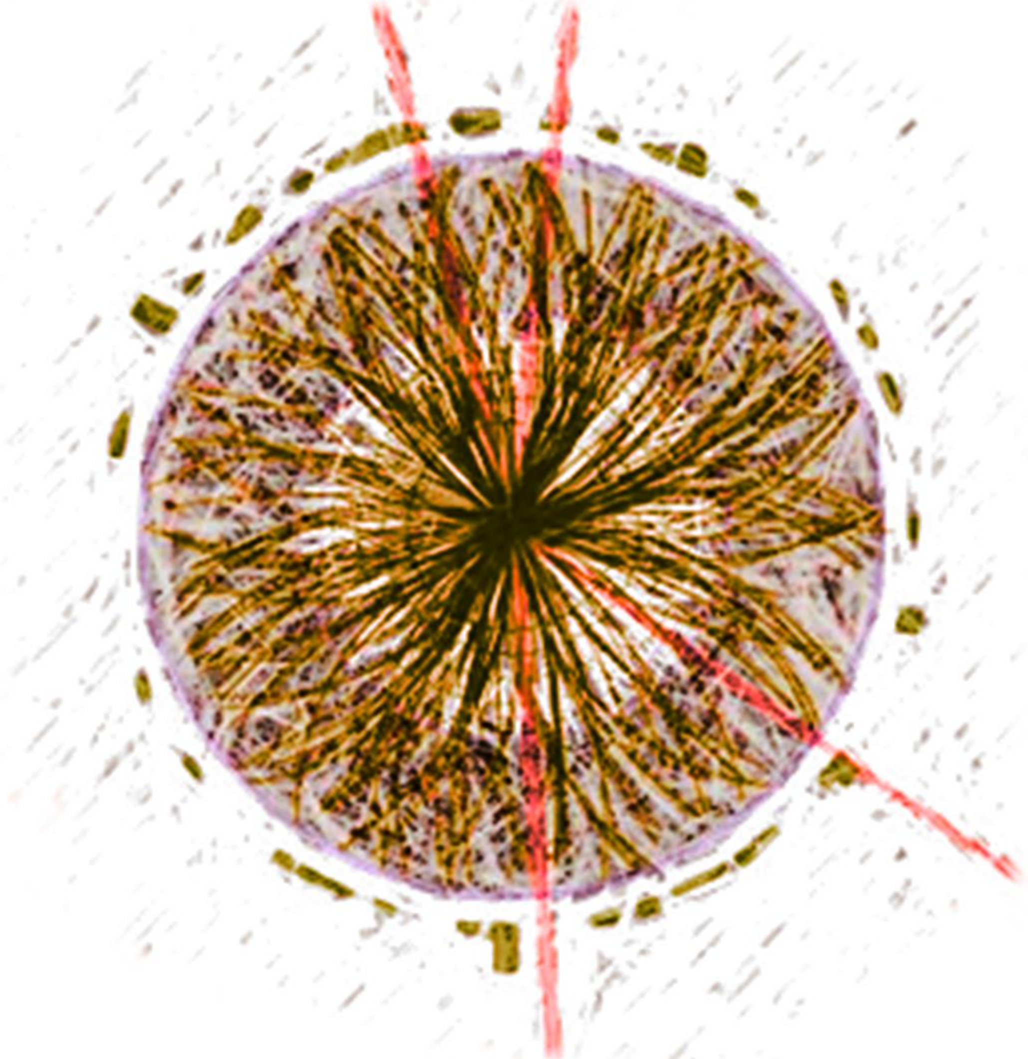
(see: `ScopeGuard ↓`)



- *C++ and Beyond 2012 http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Andrei-Alexandrescu-Systematic-Error-Handling-in-C*

- Don't worry…

  Andrei Alexandrescu says that this is perfectly fine!



The nature of Exceptions

The nature of Exceptions

- The standard has a hierarchy of exception types.
- Whilst some have questioned the utility of the hierarchy, this codification of the **reason** for the exception flow is interesting.
- There is no current analog of this for the forward flow of execution in functions.

- What would a forward equivalent of exceptions look like?

- Like exceptions:
  - Out of band (*not a parameter to every function*)
  - Inspectable
  - Capturable
  - Transportable

- But what()?

Norms?

- If functions are too complex, could *Norms* capture something about functions that we could reflect on?

Norms?

Intentions

- What we want to reify is the *intent* of programs.
- Intentions provide the context in which exceptions make sense.
- Exceptions express "*what went wrong*" in the context of "*what I was trying to do*".

- *cause* = *functions implementing* *intent*
- *effect* = *values* **or** *exceptions*

Part III

Genesis of Intent

Genesis of Intent

- Simplifying error message creation.

- The problem…

Error messages

Breakfast

```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception());
  }
}
```

```
void breakfast(recipe &fav) {
  prepare(fav);
}
```

Breakfast

```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception());
  }
}
```

```
void breakfast(recipe &fav) {
  prepare(fav);
}

void prepare(recipe &r) {
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}
```

Breakfast

```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception());
  }
}
```

```
void breakfast(recipe &fav) {
  prepare(fav);
}

void prepare(recipe &r) {
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  cupboard.get(i);
}
```

Breakfast

```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception());
  }
}
```

Breakfast

```
void breakfast(recipe &fav) {
  prepare(fav);
}

void prepare(recipe &r) {
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  cupboard.get(i);
}

void cupboard::get(ingredient &i) {
  if (empty()) {
    throw std::runtime_exception("the cupboard was bare");
  }
}
```

```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception());
  }
}
```

```
void breakfast(recipe &fav) {
  prepare(fav);
}

void prepare(recipe &r) {
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  cupboard.get(i);
}

void cupboard::get(ingredient &i) {
  if (empty()) {
    throw std::runtime_exception("the cupboard was bare");
  }
}
```

Breakfast


```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception());
  }
}
```

Breakfast

```
void breakfast(recipe &fav) {
  prepare(fav);
}

void prepare(recipe &r) {
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  cupboard.get(i);
}

void cupboard::get(ingredient &i) {
  if (empty()) {
    throw std::runtime_exception("the cupboard was bare");
  }
}
```

“ **the cupboard was bare** ”

```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception());
  }
}
```

- *“Exceptions are associated only with root reasons, not goals.*

  'I/O error' doesn’t describe 'saving weight file'.”

  Andrei Alexandrescu

“ `the cupboard was bare` ”

- Trying again, with nested exceptions…

Error messages

```
void breakfast(recipe &fav) {
  prepare(fav);
}

void prepare(recipe &r) {
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  cupboard.get( i );
}
```

```cpp
void breakfast(recipe &fav) {
  prepare(fav);
}

void prepare(recipe &r) {
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}
```

```cpp
void fetch(ingredient &i) {
  try {
    cupboard.get( i );
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not fetch ingredient: " + i));
  }
}
```

```
void breakfast(recipe &fav) {
  prepare(fav);
}
```

```
void prepare(recipe &r) {
  try {
    for(const auto &i : r.ingredients()) {
      fetch(i);
    }
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not prepare recipe: " + r));
  }
}
```

```
void fetch(ingredient &i) {
  try {
    cupboard.get( i );
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not fetch ingredient: " + i));
  }
}
```

```
void breakfast(recipe &fav) {
  try {
    prepare(fav);
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not have breakfast”));
  }
}

void prepare(recipe &r) {
  try {
    for(const auto &i : r.ingredients()) {
      fetch(i);
    }
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not prepare recipe: " + r));
  }
}

void fetch(ingredient &i) {
  try {
    cupboard.get( i );
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not fetch ingredient: " + i));
  }
}
```

```
void breakfast(recipe &fav) {
  try {
    prepare(fav);
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not have breakfast”));
  }
}

void prepare(recipe &r) {
  try {
    for(const auto &i : r.ingredients()) {
      fetch(i);
    }
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not prepare recipe: " + r));
  }
}
```

the cupboard was bare

```
void fetch(ingredient &i) {
  try {
    cupboard.get( i );
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not fetch ingredient: " + i));
  }
}
```

```
void breakfast(recipe &fav) {
  try {
    prepare(fav);
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not have breakfast”));
  }
}

void prepare(recipe &r) {
  try {
    for(const auto &i : r.ingredients()) {
      fetch(i);
    }
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not prepare recipe: " + r));
  }
}

void fetch(ingredient &i) {
  try {
    cupboard.get( i );
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not fetch ingredient: " + i));
  }
}
```

the cupboard was bare

```cpp
void breakfast(recipe &fav) {
  try {
    prepare(fav);
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not have breakfast”));
  }
}

void prepare(recipe &r) {
  try {
    for(const auto &i : r.ingredients()) {
      fetch(i);
    }
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not prepare recipe: " + r));
  }
}

void fetch(ingredient &i) {
  try {
    cupboard.get( i );
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not fetch ingredient: " + i));
  }
}
```

the cupboard was bare

```
void breakfast(recipe &fav) {
  try {
    prepare(fav);
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not have breakfast”));
  }
}

void prepare(recipe &r) {
  try {
    for(const auto &i : r.ingredients()) {
      fetch(i);
    }
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not prepare recipe: " + r));
  }
}

void fetch(ingredient &i) {
  try {
    cupboard.get( i );
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not fetch ingredient: " + i));
  }
}
```

the cupboard was bare

```
void breakfast(recipe &fav) {
  try {
    prepare(fav);
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could
  }
}

void prepare(recipe &r) {
  try {
    for(const auto &i : r.ingredients()) {
      fetch(i);
    }
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not prepare recipe: " + r));
  }
}

void fetch(ingredient &i) {
  try {
    cupboard.get( i );
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not fetch ingredient: " + i));
  }
}
```

“
```
could not have breakfast
 could not prepare recipe: bacon and eggs
  could not fetch ingredient: eggs
   the cupboard was bare
```
”

the cupboard was bare

```
void breakfast(recipe &fav) {
  try {
    prepare(fav);
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not have breakfast"));
  }
}

void prepare(recipe &r) {
  try {
    for(const auto &i : r.ingredients()) {
      fetch(i);
    }
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not prepare recipe: " + r));
  }
}

void fetch(ingredient &i) {
  try {
    cupboard.get( i );
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not fetch ingredient: " + i));
  }
}
```

```
void breakfast(recipe &fav) {

    prepare(fav);



}

void prepare(recipe &r) {

    for(const auto &i : r.ingredients()) {
      fetch(i);
    }



}

void fetch(ingredient &i) {

    cupboard.get( i );



}
```

A dog’s breakfast

```
void breakfast(recipe &fav) {
  try {
    prepare(fav);
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not have breakfast”));
  }
}

void prepare(recipe &r) {
  try {
    for(const auto &i : r.ingredients()) {
      fetch(i);
    }
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not prepare recipe: " + r));
  }
}

void fetch(ingredient &i) {
  try {
    cupboard.get( i );
  } catch(...) {
    std::throw_with_nested(std::runtime_error("could not fetch ingredient: " + i));
  }
}
```

- A gedanken experiment…

Error messages

```
void breakfast(recipe &fav) {
  prepare(fav);
}

void prepare(recipe &r) {
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  cupboard.get(i);
}
```

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

the cupboard was bare

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

the cupboard was bare

```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception(),
          current_intentions());
  }
}
```

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

“ **whilst** having breakfast
whilst preparing ***bacon and eggs***
whilst fetching ***eggs***
the cupboard was bare ”


```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception(),
          current_intentions());
  }
}
```

```cpp
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

“ **whilst** having breakfast
 whilst preparing ***bacon and eggs***
  whilst fetching ***eggs***
   the cupboard was bare ”

We expressed intent

```cpp
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception(),
          current_intentions());
  }
}
```

- *“Exceptions are associated only with root reasons, not **goals**.*

  'I/O error' doesn’t describe 'saving weight file'.”

  Andrei Alexandrescu

“
```
whilst having breakfast
 whilst preparing bacon and eggs
  whilst fetching eggs
   the cupboard was bare
```
”

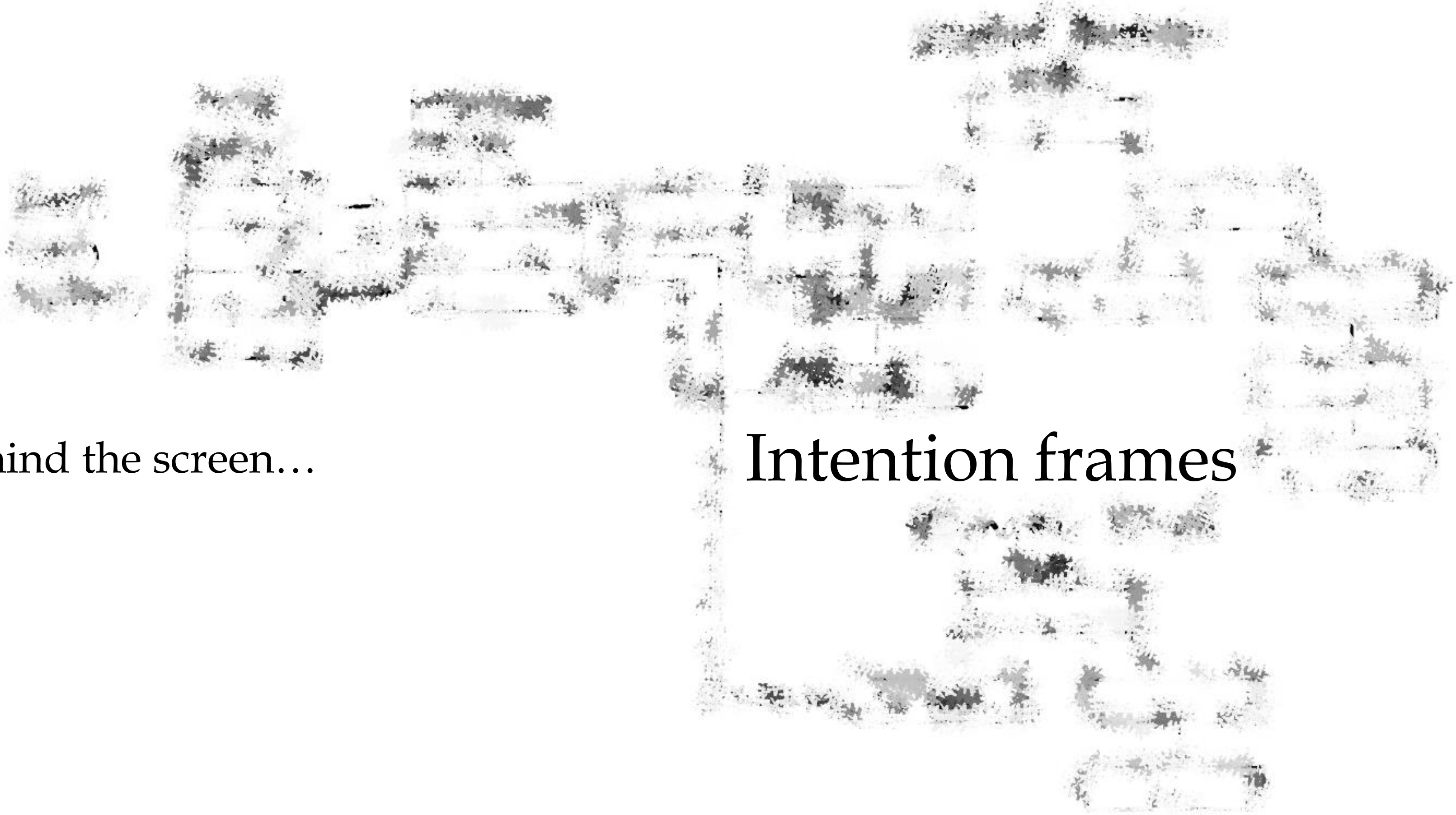
```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

“**whilst** having breakfast
whilst preparing ***bacon and eggs***
whilst fetching ***eggs***
the cupboard was bare”

Expressing intent

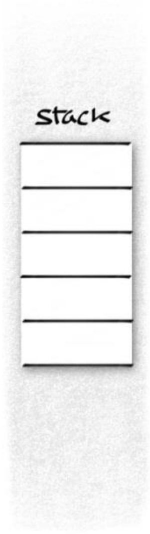
```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception(),
          current_intentions());
  }
}
```

Intention frames

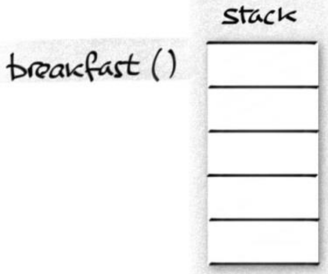


- Behind the screen…

An unintentional breakfast

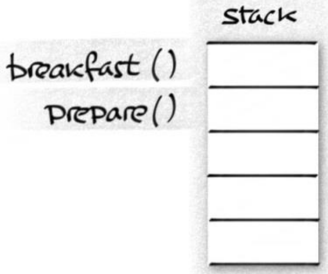


breakfast



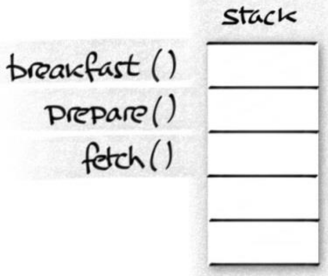
breakfast

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}
```



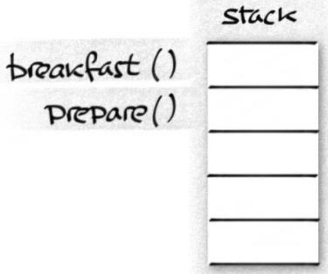
breakfast

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}
```



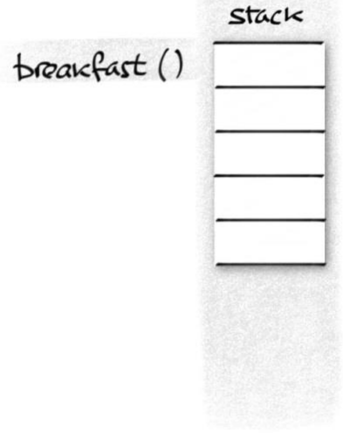
breakfast

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```



breakfast

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}
```

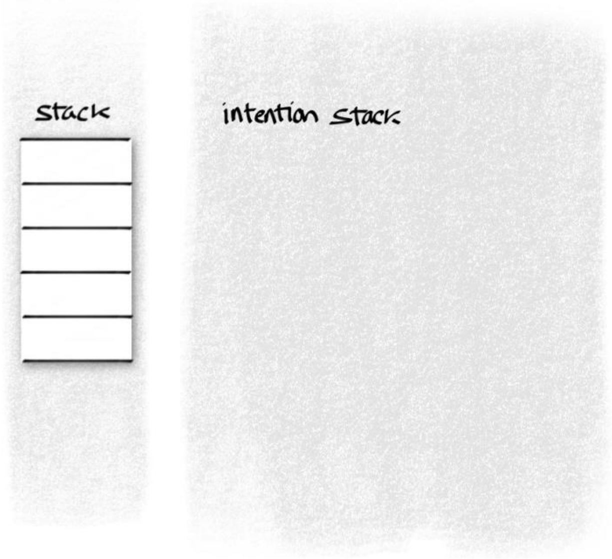


breakfast

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}
```

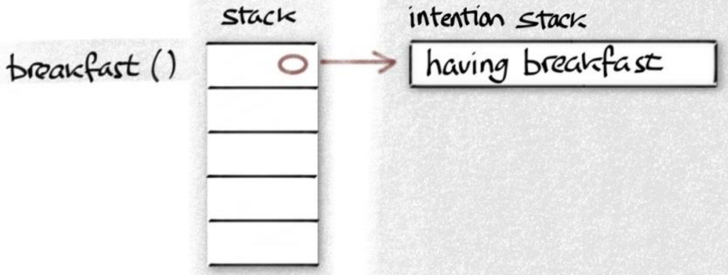
breakfast

An
intentional
breakfast



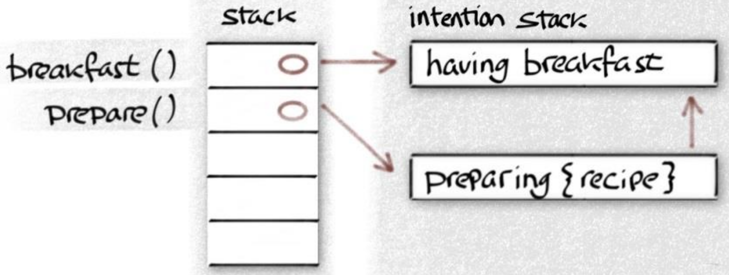
breakfast

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```



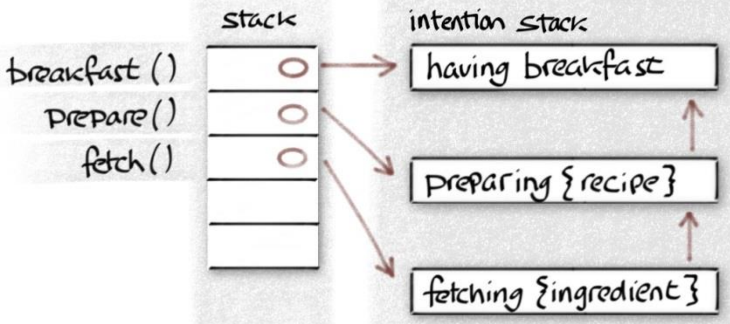
breakfast

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```



breakfast

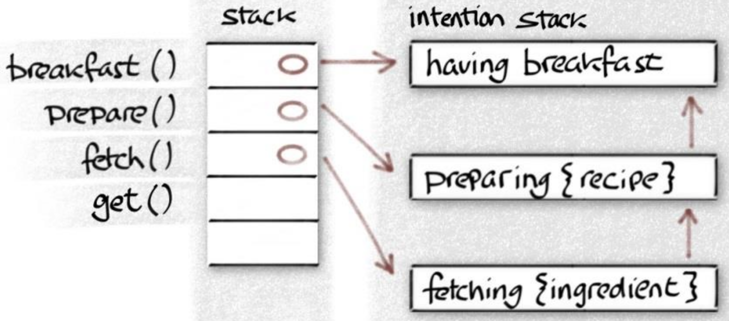
```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```



breakfast

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

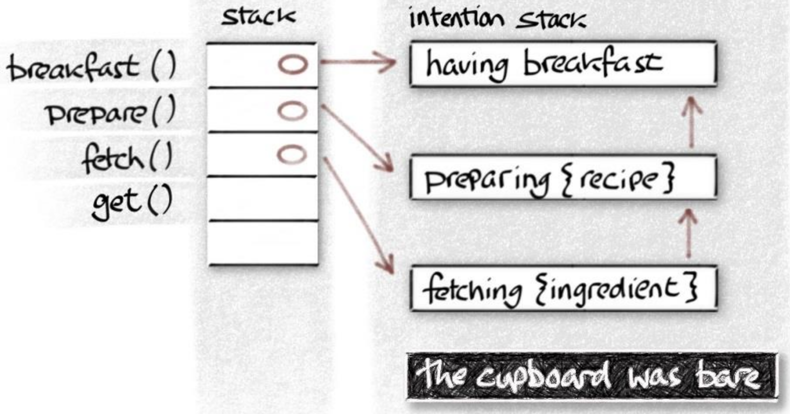
breakfast




```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

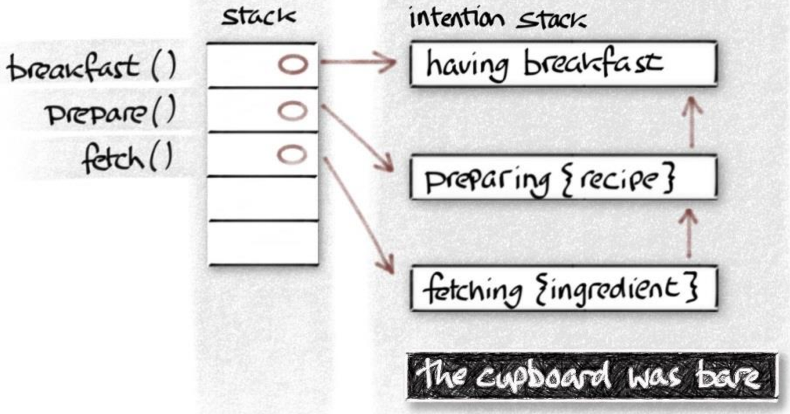
```
void cupboard::get(ingredient &i) {
  if (empty()) {
    throw std::runtime_exception("the cupboard was bare");
  }
}
```

breakfast




```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

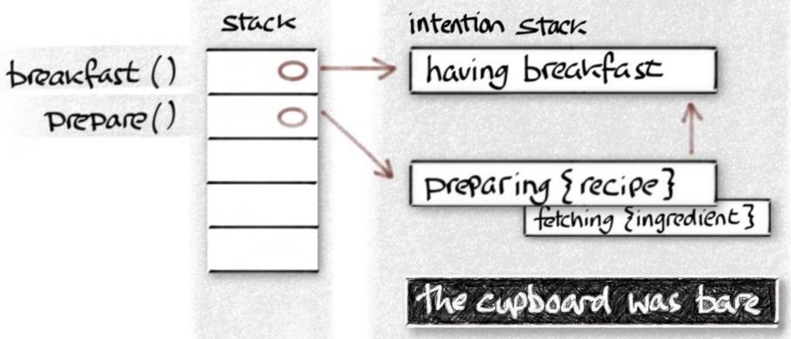
```
void cupboard::get(ingredient &i) {
  if (empty()) {
    throw std::runtime_exception("the cupboard was bare");
  }
}
```



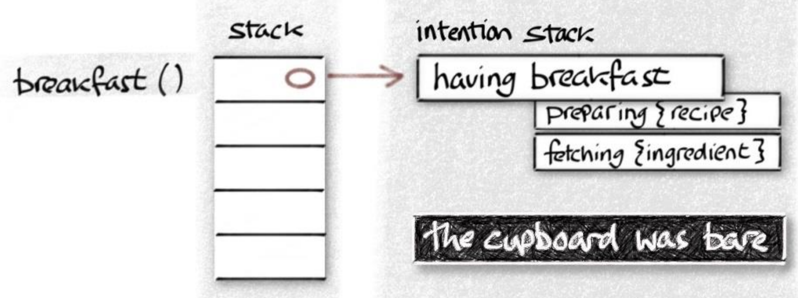
breakfast

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

breakfast



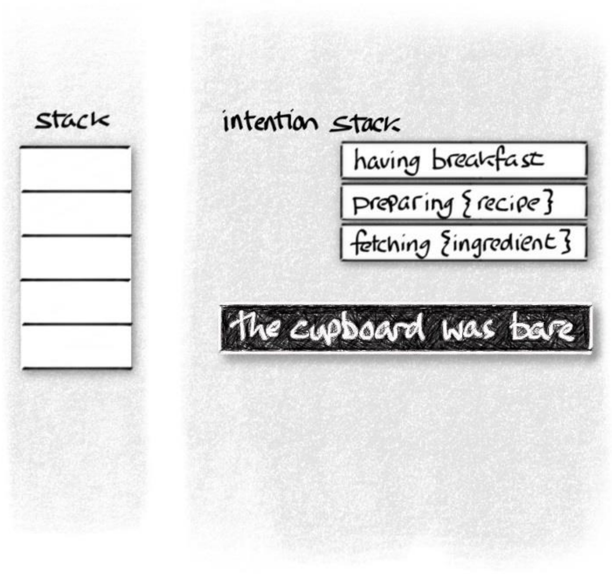

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}
```



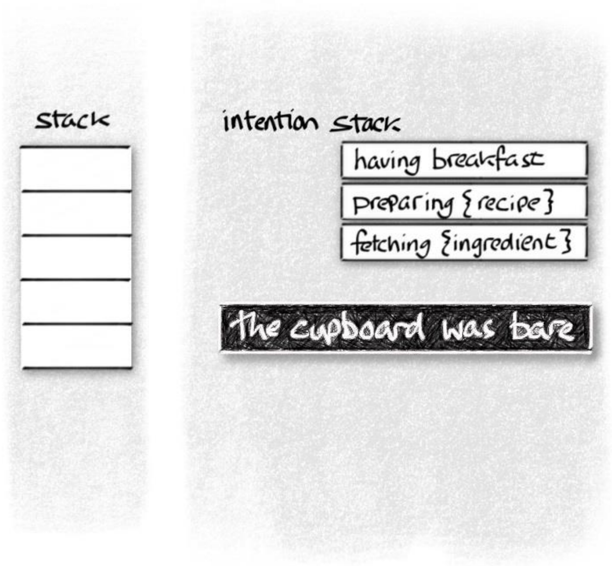
breakfast

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}
```

breakfast




```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception(),
          current_intentions());
  }
}
```

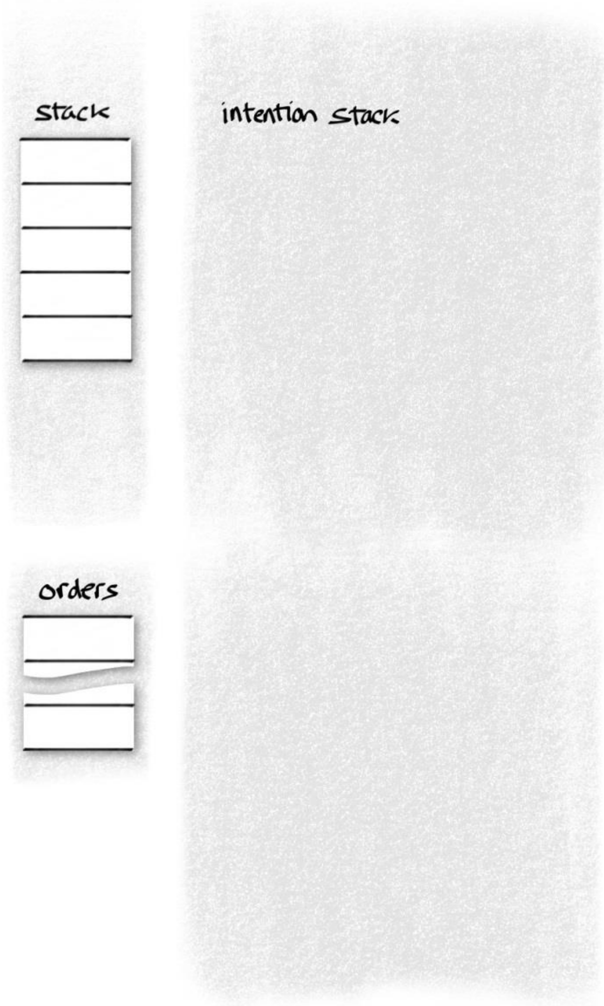


understanding our disappointment

“ **whilst** having breakfast
 whilst preparing ***bacon and eggs***
  whilst fetching ***eggs***
   <u>*the cupboard was bare*</u> ”

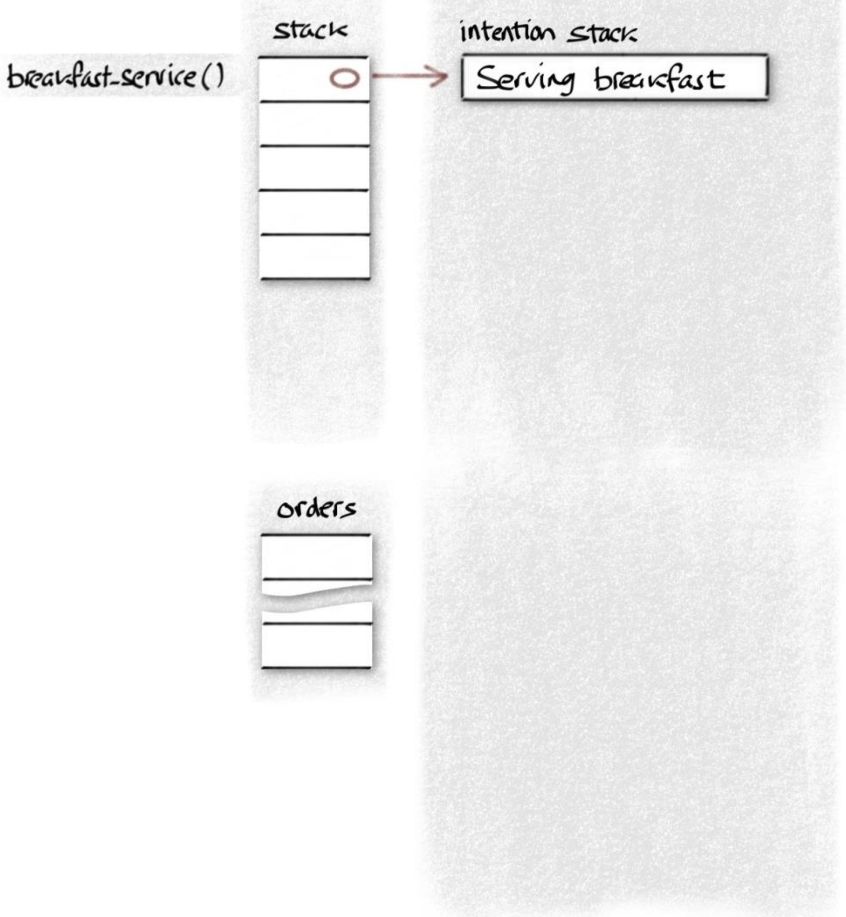
```
void main() {
  try {
    breakfast(bacon_and_eggs);
  } catch(...) {
    error(std::current_exception(),
          current_intentions());
  }
}
```

An
exceptional
Cafe



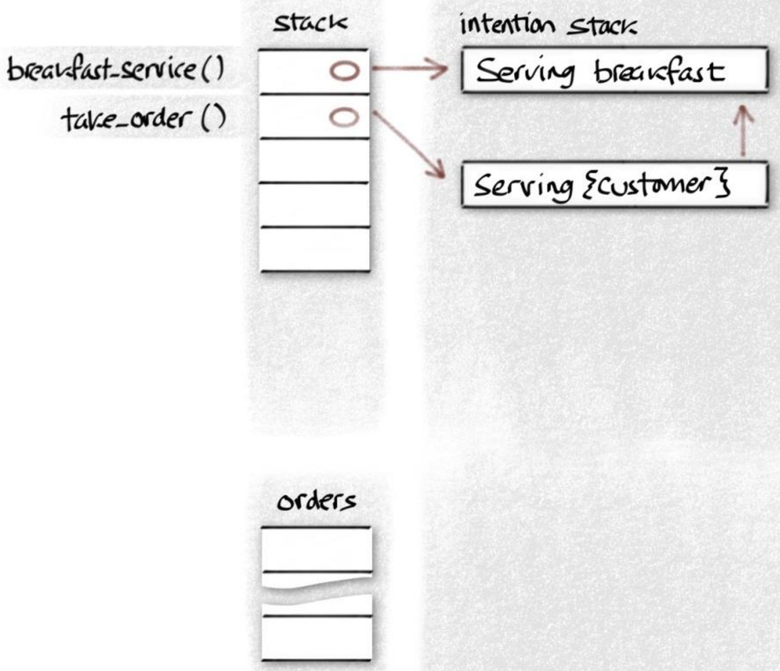
the cafe

```
void breakfast_service() {
  whilst("serving breakfast");
  while (customers.waiting())
    take_order(customers.dequeue());
  }
}

void take_order(customer c) {
  whilst("serving {customer}", c);
  orders.queue(order(c,
                     c.choice(),
                     current_intentions()));
}
```



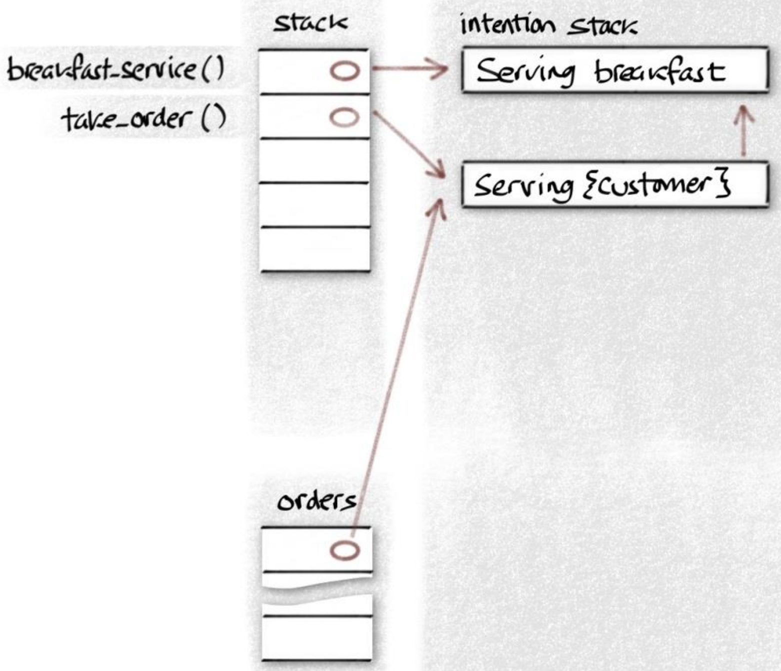
the cafe

```
void breakfast_service() {
  whilst("serving breakfast");
  while (customers.waiting())
    take_order(customers.dequeue());
  }
}

void take_order(customer c) {
  whilst("serving {customer}", c);
  orders.queue(order(c,
                     c.choice(),
                     current_intentions()));
}
```



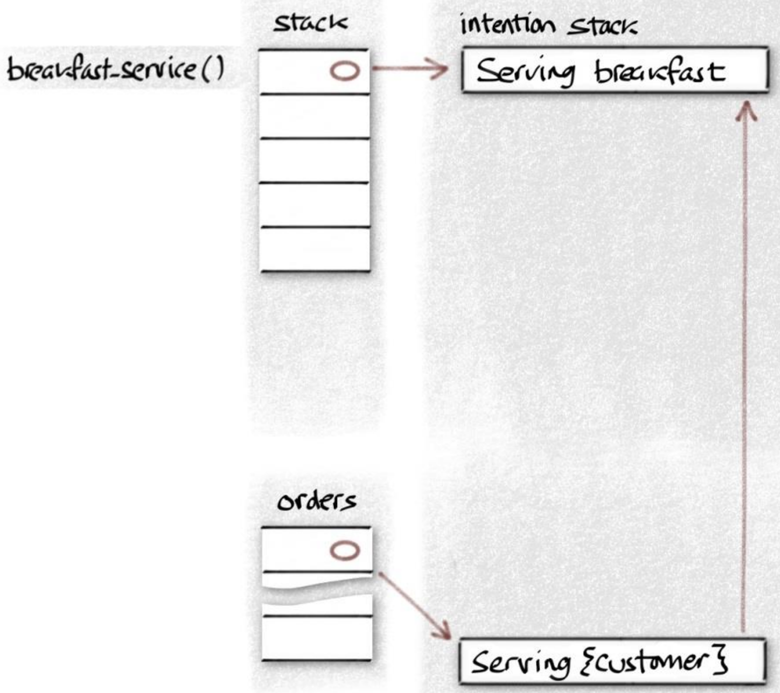
the cafe

```
void breakfast_service() {
  whilst("serving breakfast");
  while (customers.waiting())
    take_order(customers.dequeue());
  }
}

void take_order(customer c) {
  whilst("serving {customer}", c);
  orders.queue(order(c,
                     c.choice(),
                     current_intentions()));
}
```



the cafe

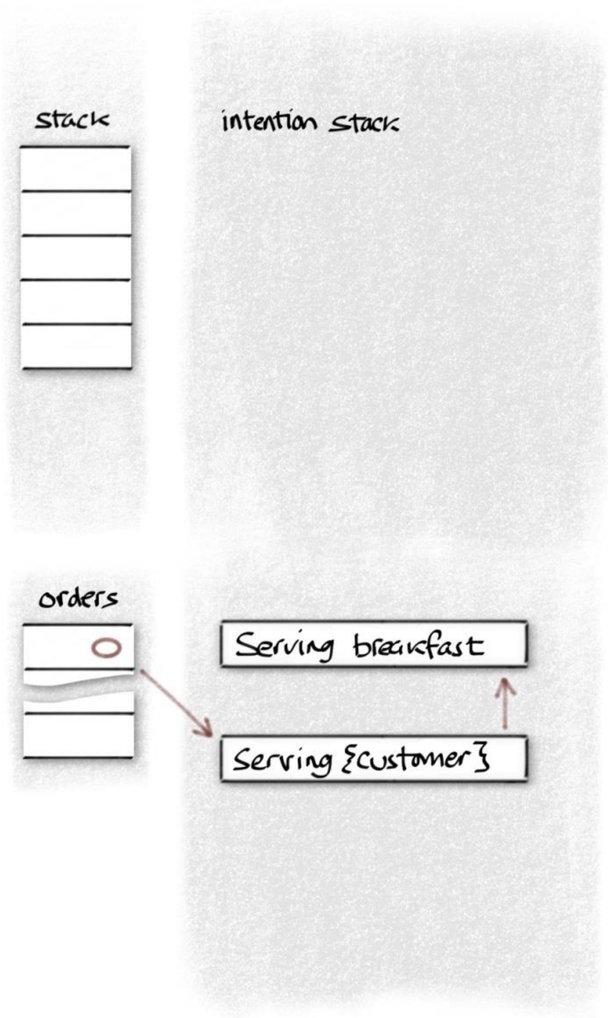
```
void breakfast_service() {
  whilst("serving breakfast");
  while (customers.waiting())
    take_order(customers.dequeue());
  }
}

void take_order(customer c) {
  whilst("serving {customer}", c);
  orders.queue(order(c,
                     c.choice(),
                     current_intentions()));
}
```

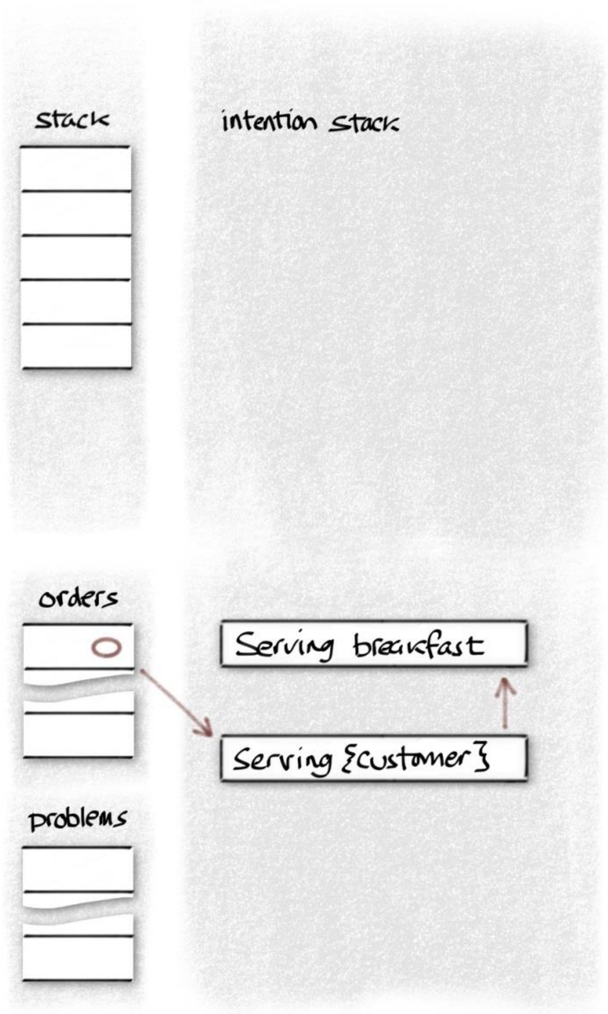


the cafe

```
void breakfast_service() {
  whilst("serving breakfast");
  while (customers.waiting())
    take_order(customers.dequeue());
  }
}

void take_order(customer c) {
  whilst("serving {customer}", c);
  orders.queue(order(c,
                     c.choice(),
                     current_intentions()));
}
```

the kitchen

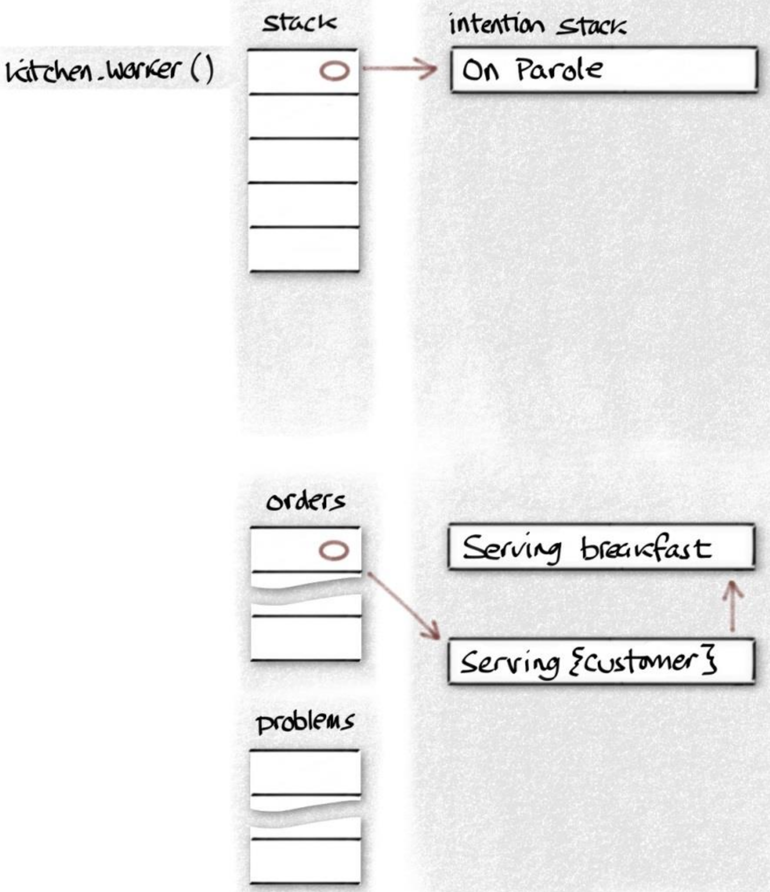

stack
intention stack
orders
Serving breakfast
Serving {customer}



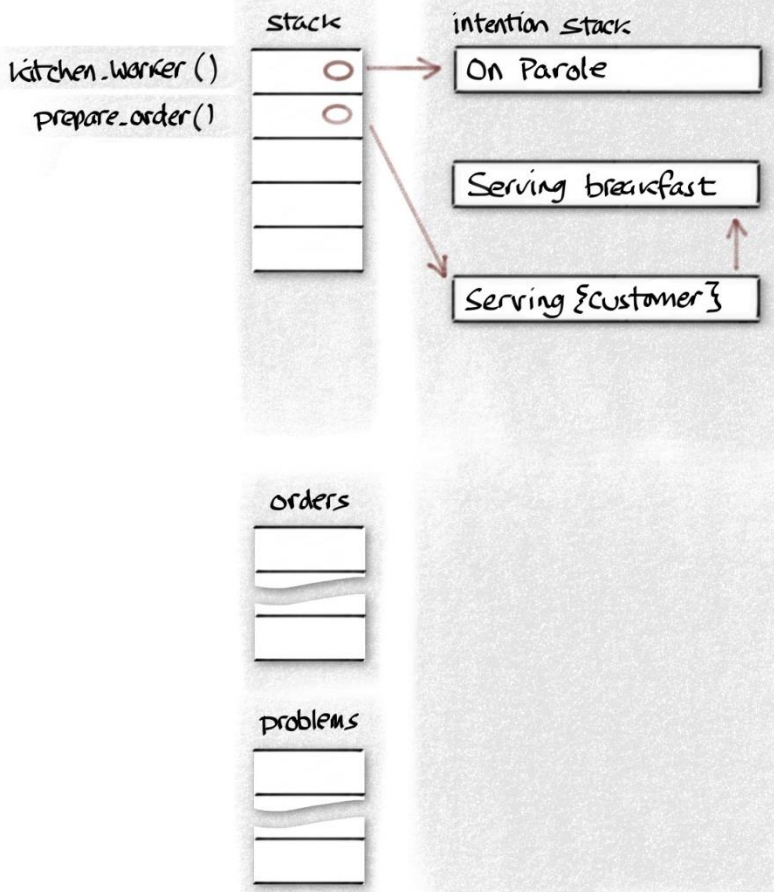
the kitchen

```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```

the kitchen

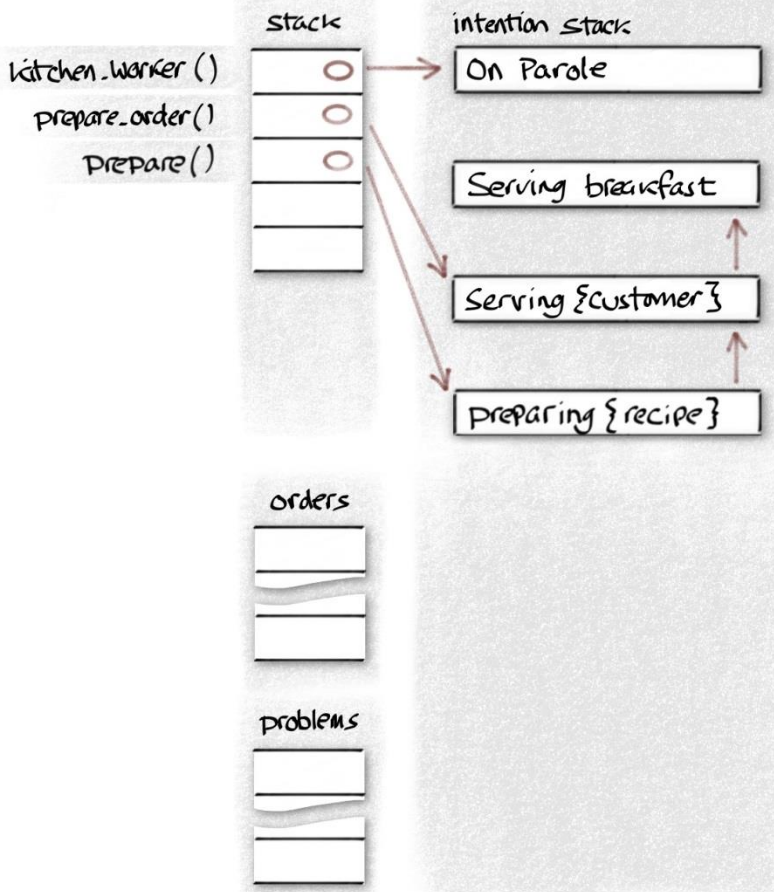



```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```



the kitchen

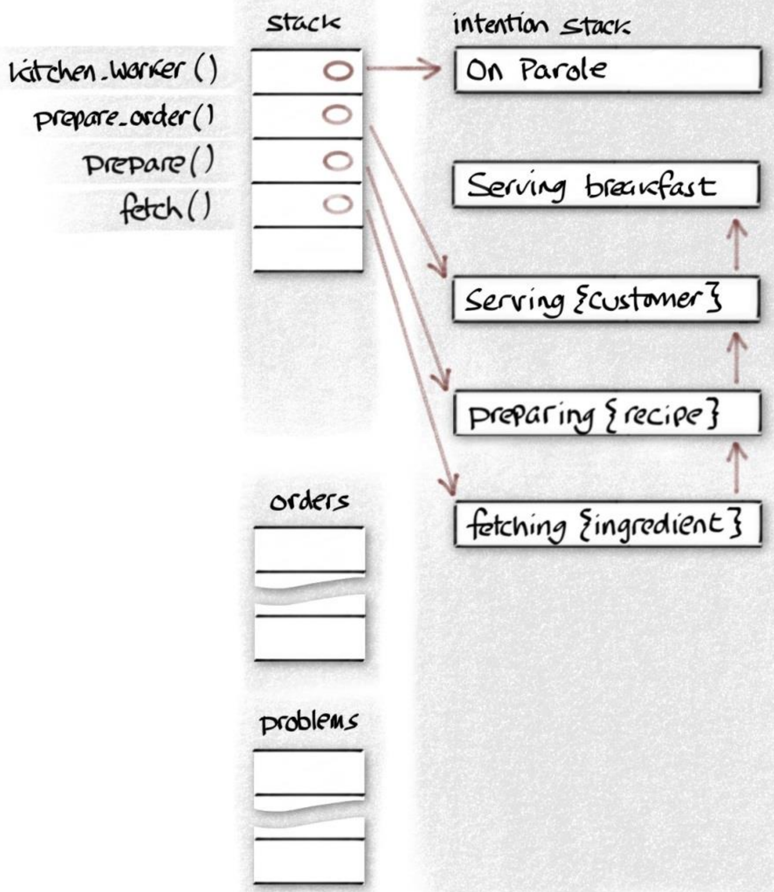
```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```



the kitchen

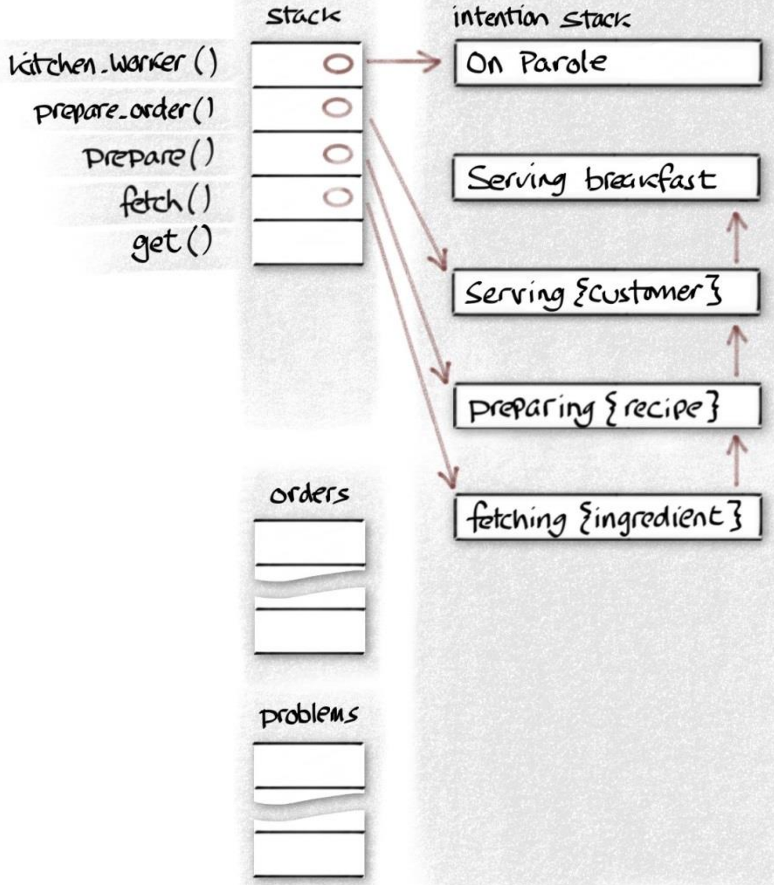
```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```

the kitchen



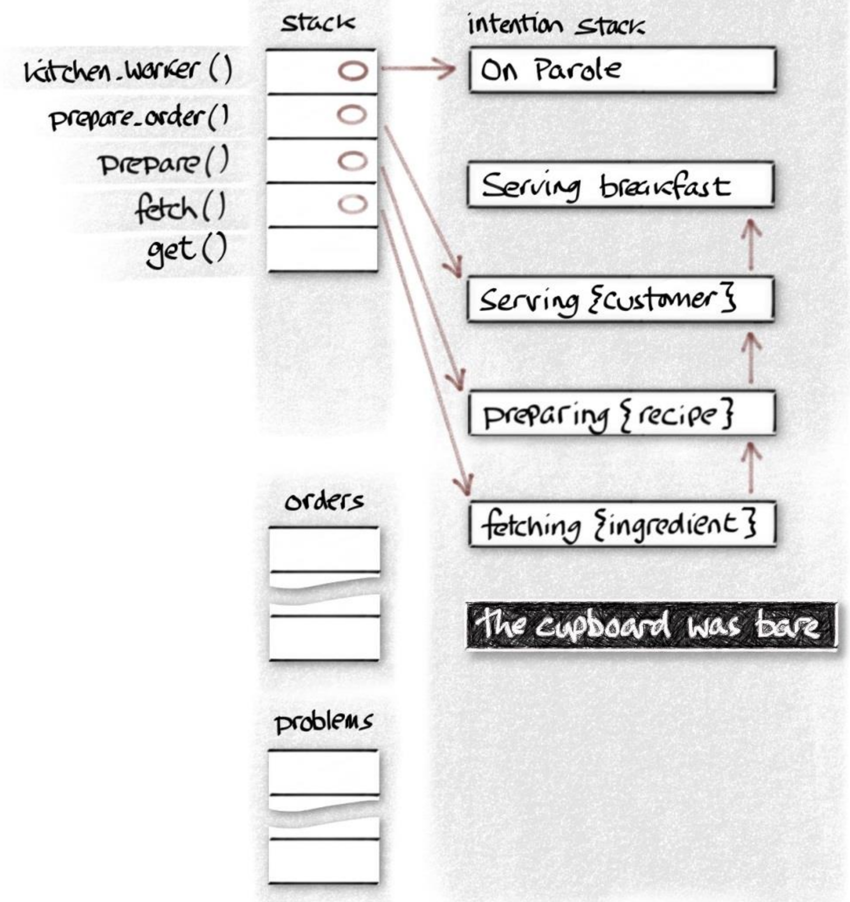

```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```

the kitchen

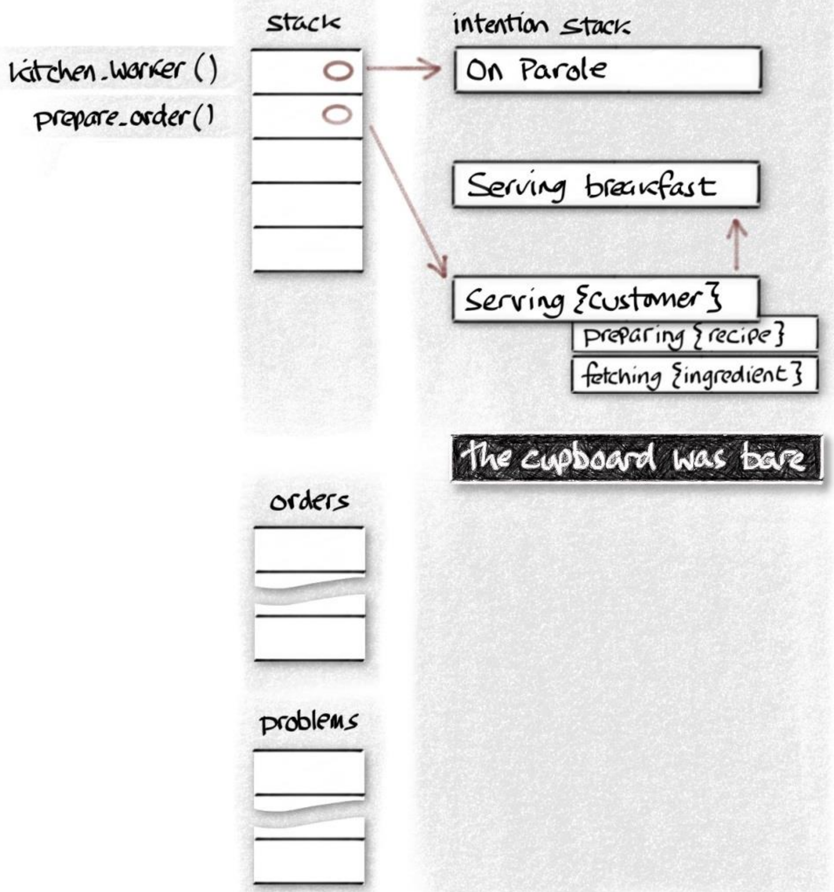



```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                          std::current_exception(),
                          current_intentions()));
  }
}
```

the kitchen



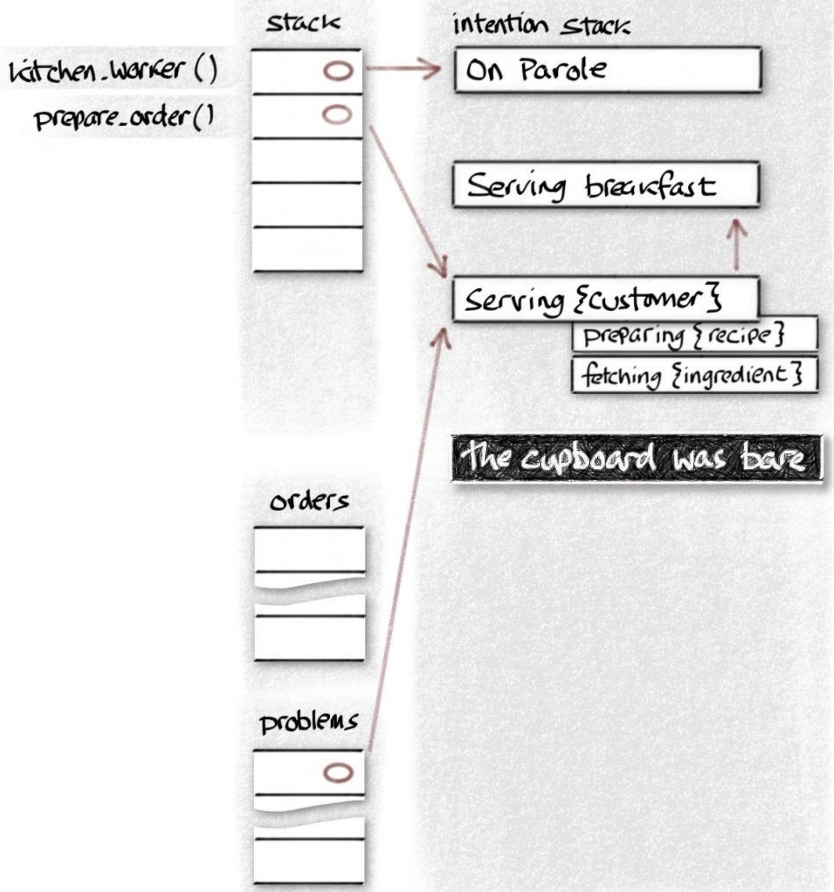

```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```



the kitchen

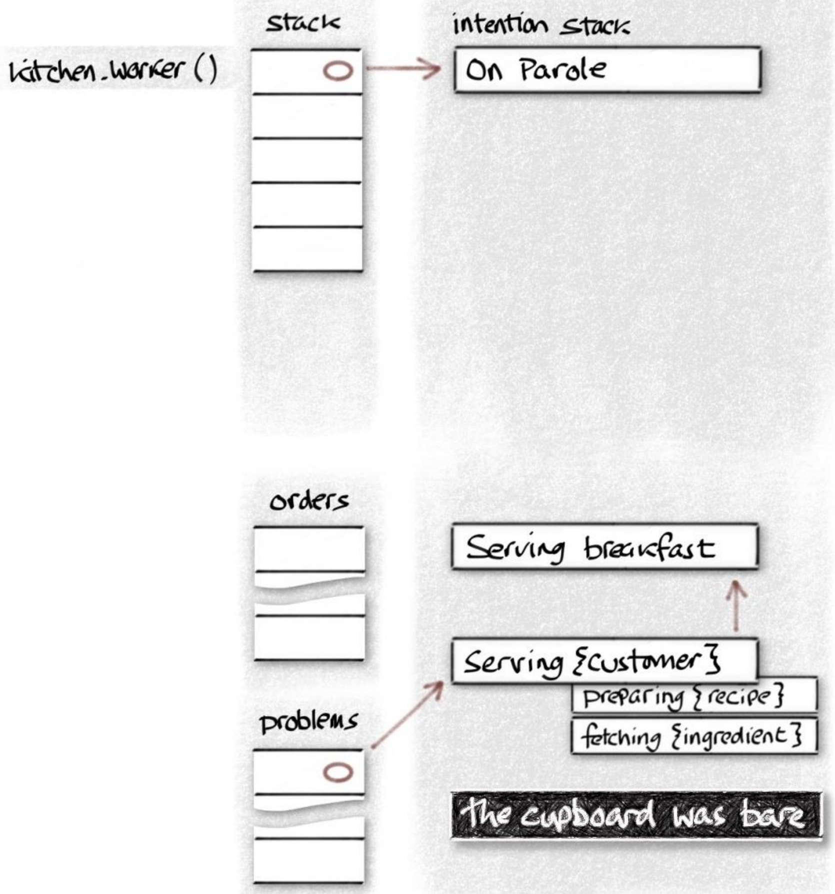
```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```

the kitchen

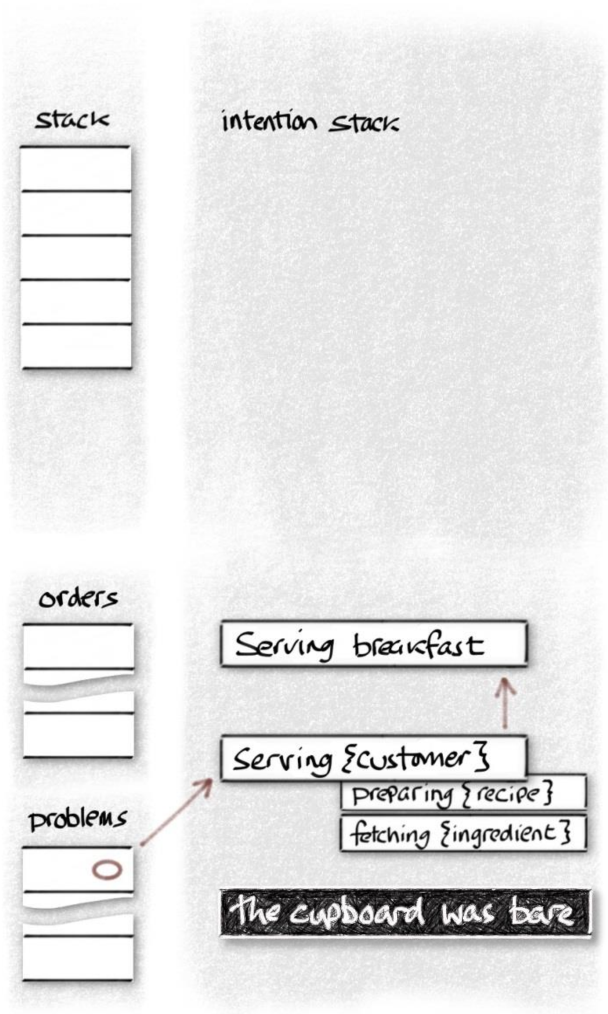



```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                          std::current_exception(),
                          current_intentions()));
  }
}
```

the kitchen

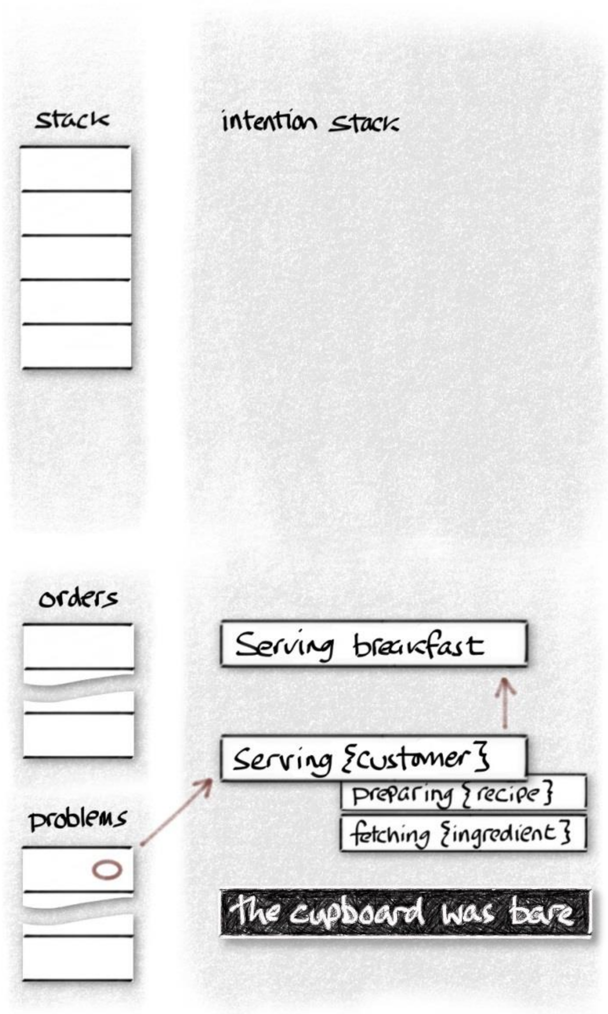



```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```



complaints dept.

?



complaints dept.

?

“ **whilst** serving breakfast
 whilst serving **{CUSTOMER}**
  whilst preparing **{RECIPE}**
   whilst fetching **{INGREDIENT}**
    {EXCEPTION}

 whilst explaining that **{ISSUE}**
  {EXCEPTION} ”

Declarative

- Declarative expression of intent
  - is more succinct
  - has fewer execution paths to test
  - is executable documentation
- … but at what cost?

What would an implementation involve?

What's in a **`whilst`**?

```
#define _PASTE_(A, B) A ## B
#define _NAME_(PREFIX, N) _PASTE_(PREFIX, N)

#define INTENTION_ID _NAME_(_intention_, __LINE__)
#define SCOPE_NAME _NAME_(_scope_, __LINE__)

#define whilst(DESC, ...) \
  static intention *INTENTION_ID = runtime::inter(__FILE__, __LINE__, DESC); \
  scope SCOPE_NAME(INTENTION_ID, values(__VA_ARGS__));
```

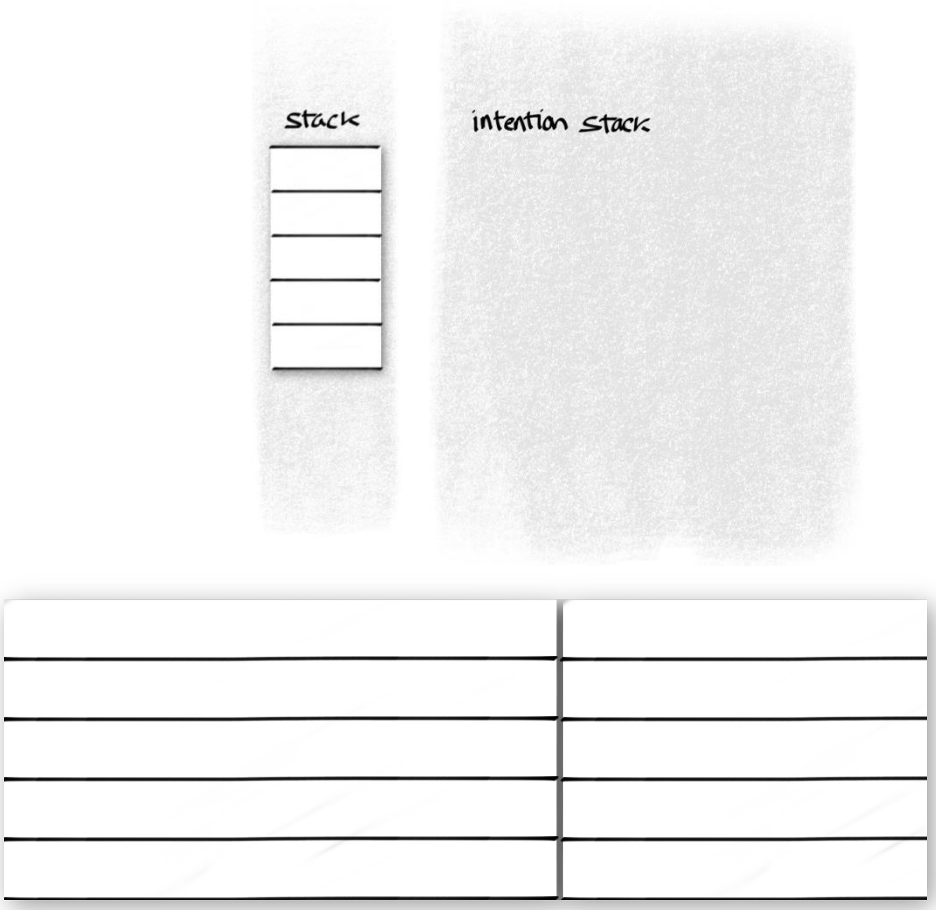
What's in a **`whilst`**?

```
whilst("preparing {recipe}", "bacon and eggs")
```

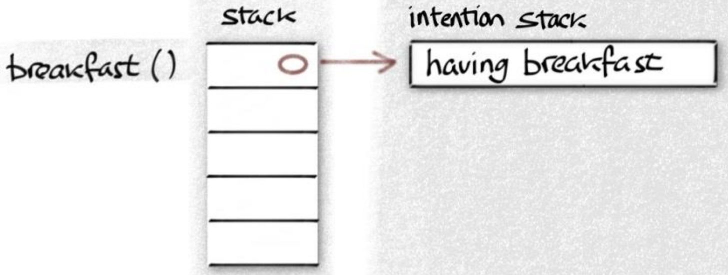
```
static intention *_intention_101 = runtime::inter("cooking.cpp", 101, "preparing {recipe}");
scope _scope_101(_intention_101, values("bacon and eggs"));
```

Interring intentions

```
static intention *_intention_101 = runtime::inter(“cooking.cpp", 101, "preparing {recipe}");
```

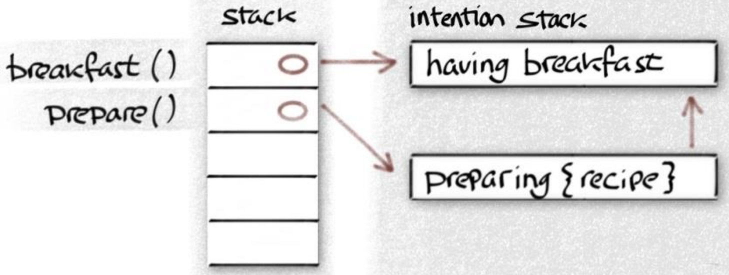



```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```



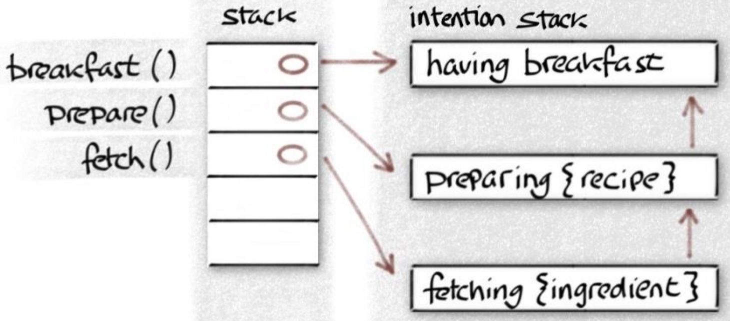
1	*having breakfast*	home.cpp	100

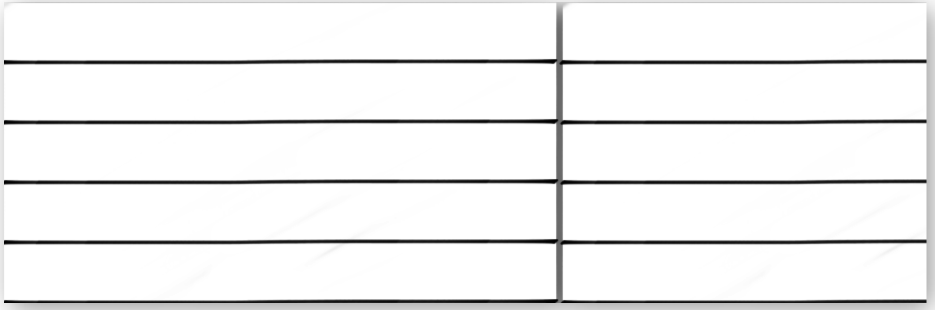
```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```



1	*having breakfast*	home.cpp	100
2	*preparing {recipe}*	cooking.cpp	101

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```





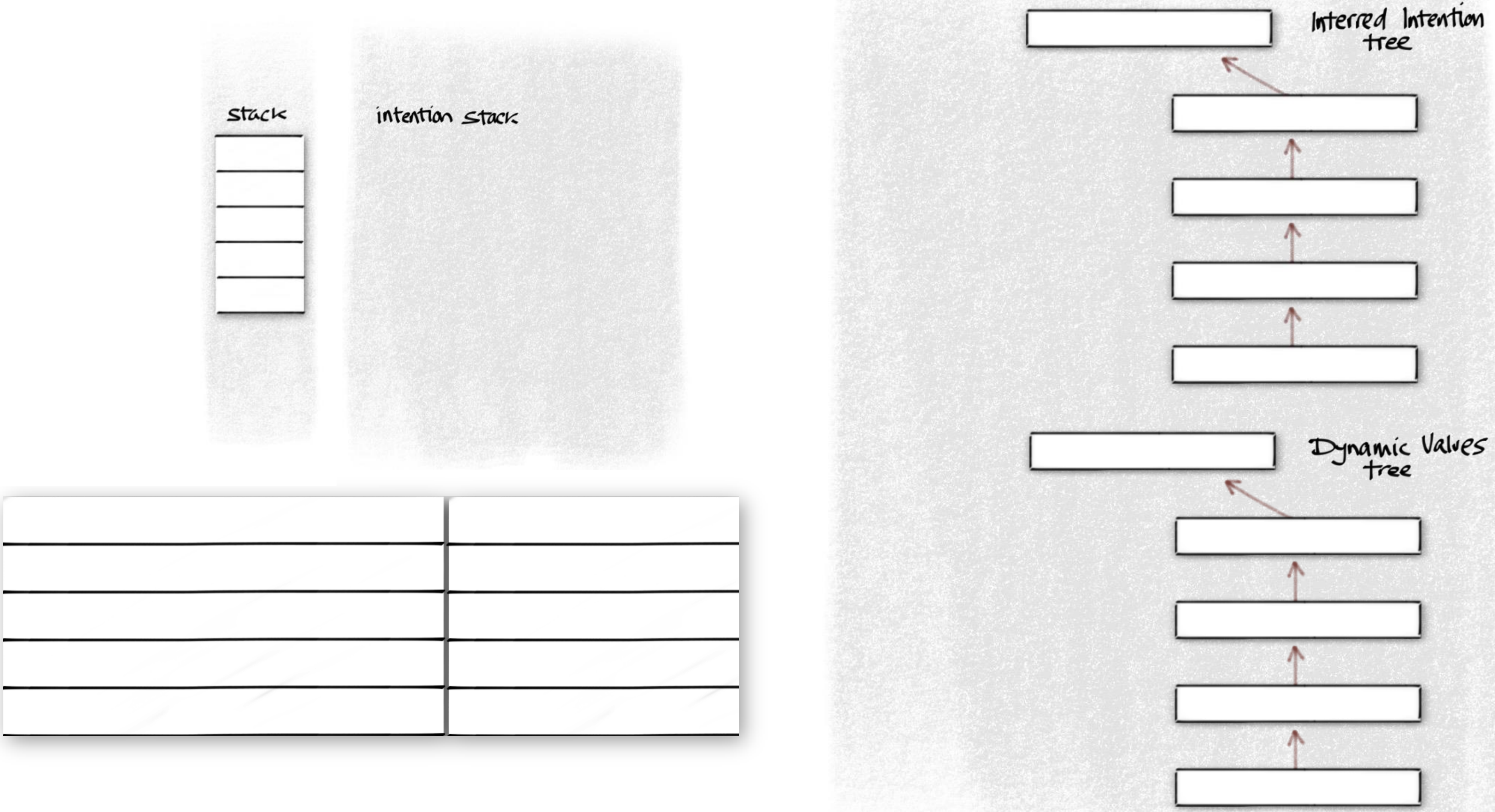
```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

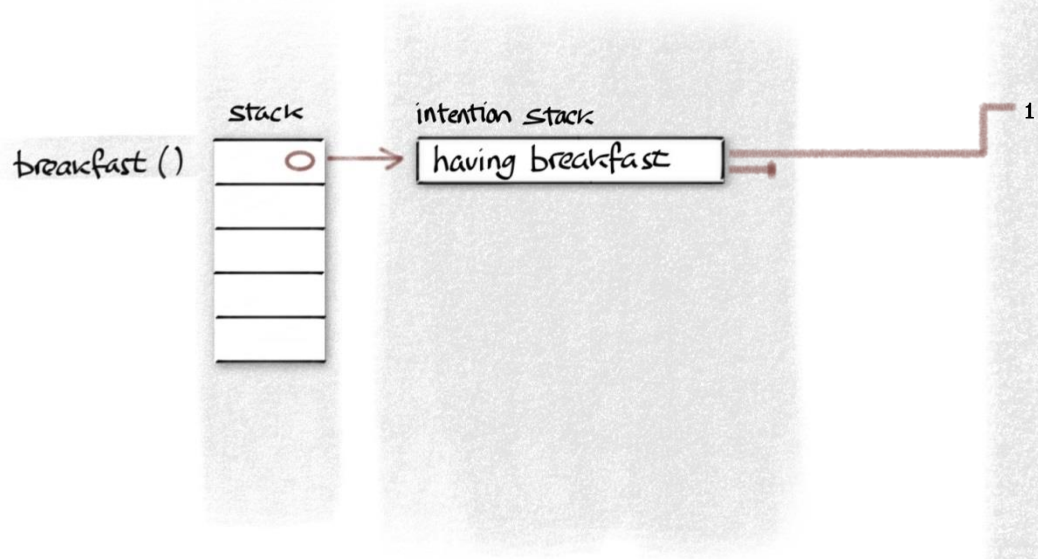
Intention scopes

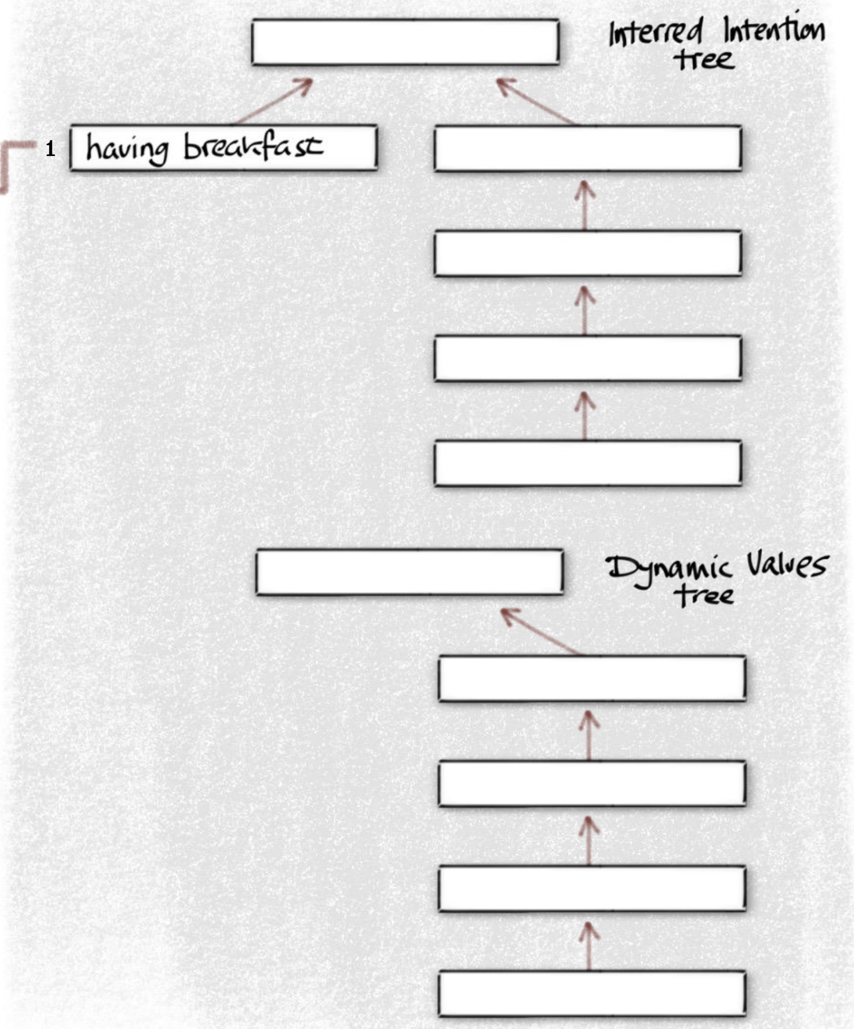
```
static intention *_intention_101 = runtime::inter(“cooking.cpp", 101, "preparing {recipe}");
scope _scope_101(_intention_101, values(“bacon and eggs"));
```

Intention scopes

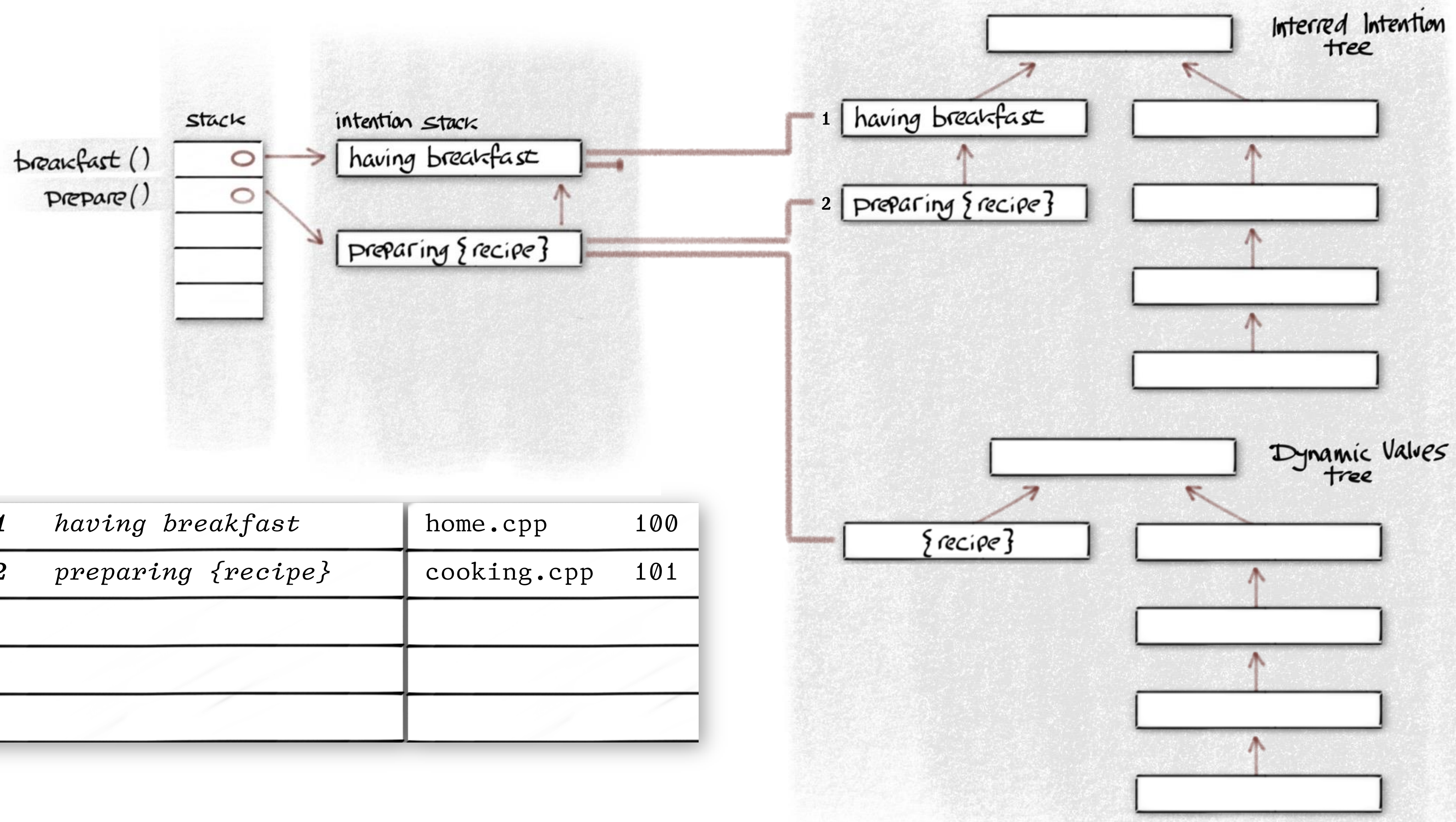
```
struct scope {
  int uncaught_;

  scope(intention *i, values &v) {
    uncaught_ = uncaught_exceptions();
    runtime.enter(i,v);
  }
  ~scope() {
    runtime.leave(uncaught_);
  }
}

void runtime::enter(intention *i, values &v) {
  push(id, v);
}

void runtime::leave(int uncaught) {
  if (uncaught_exceptions() != uncaught)
    throwing();
  pop();
}
```


stack
intention stack
Interred Intention tree
Dynamic Values tree

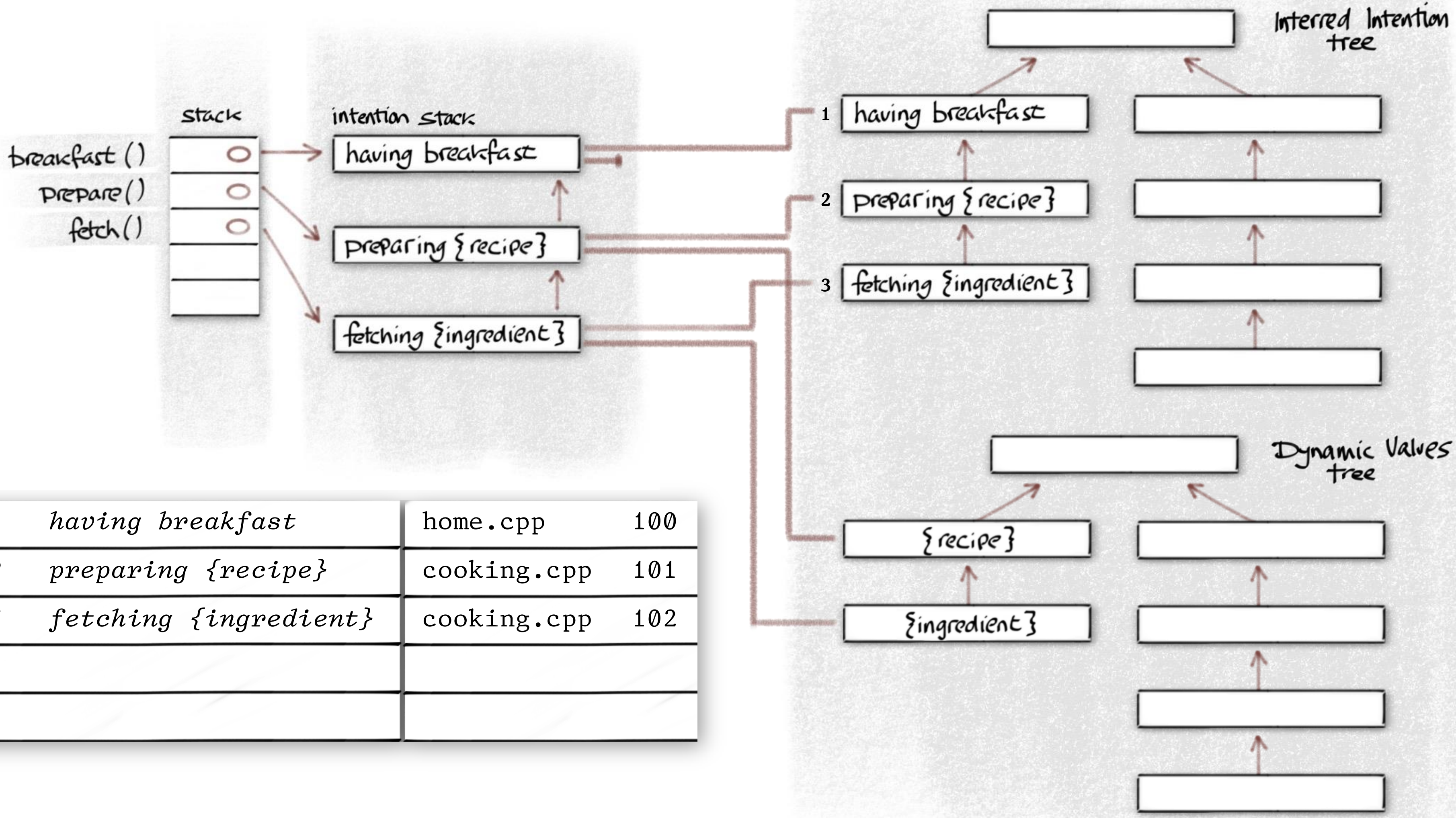




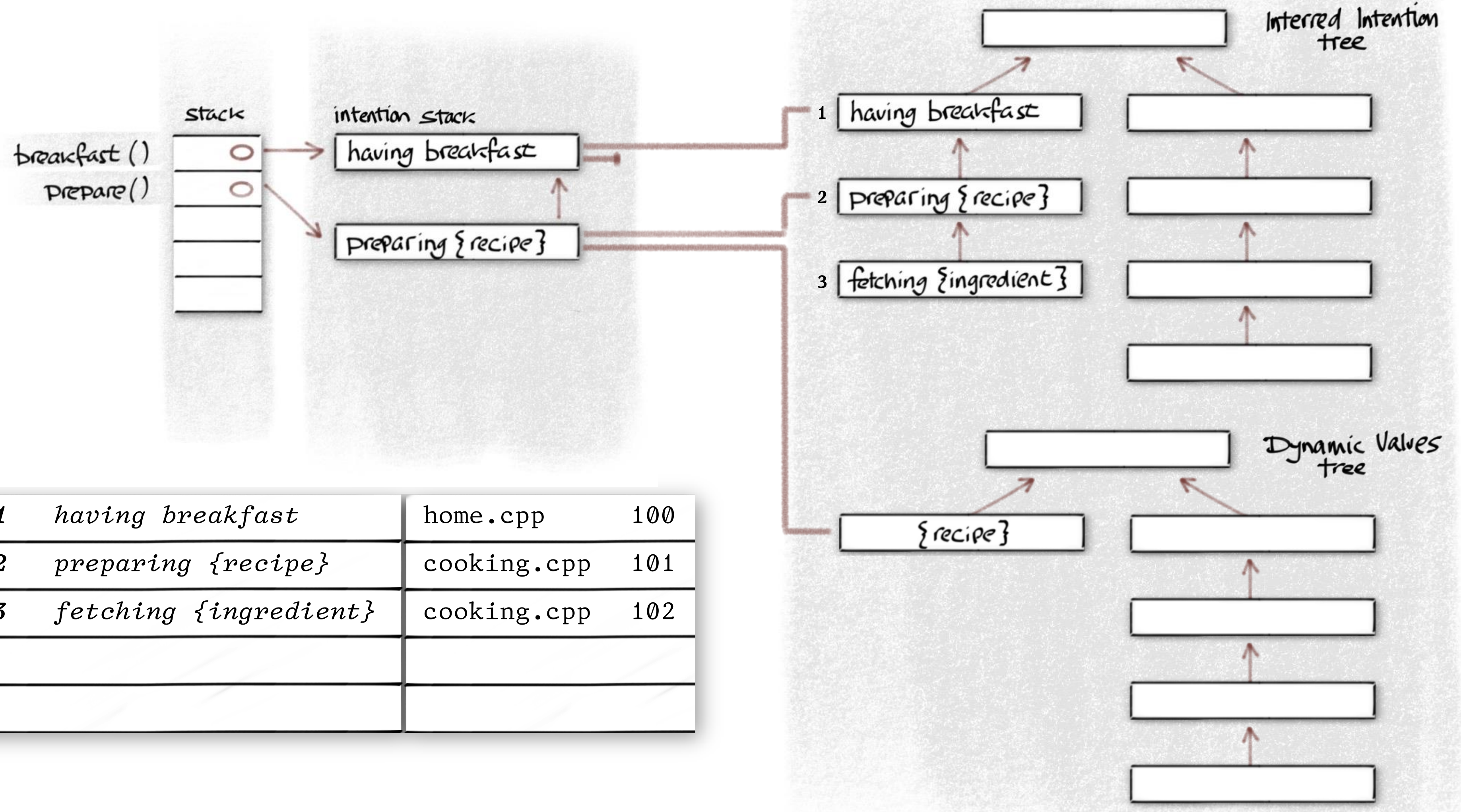
1	*having breakfast*	home.cpp	100



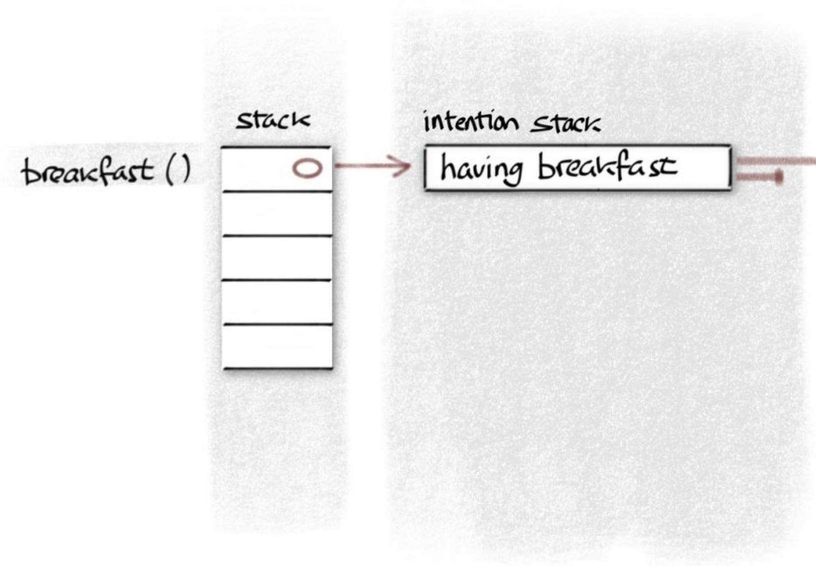
1	*having breakfast*	home.cpp	100
2	*preparing {recipe}*	cooking.cpp	101

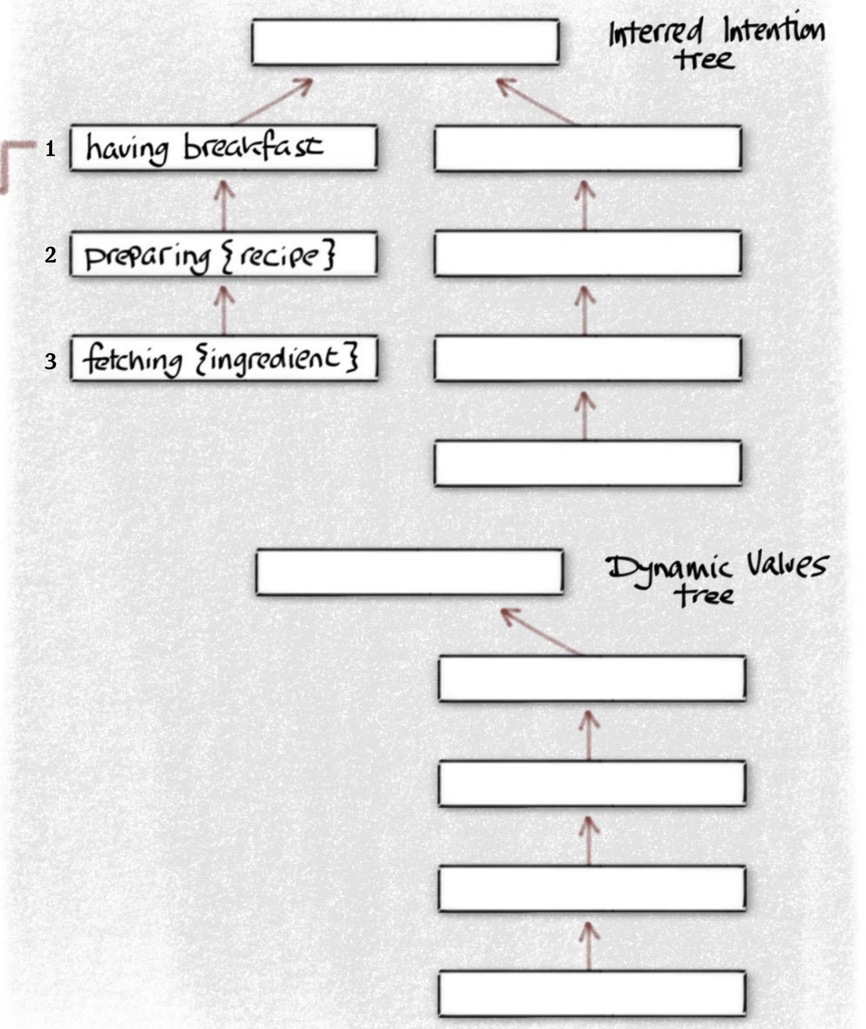


1	*having breakfast*	home.cpp	100
2	*preparing {recipe}*	cooking.cpp	101
3	*fetching {ingredient}*	cooking.cpp	102

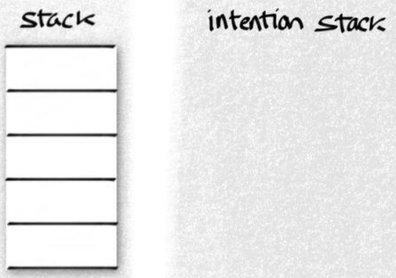


1	*having breakfast*		home.cpp	100
2	*preparing {recipe}*		cooking.cpp	101
3	*fetching {ingredient}*		cooking.cpp	102

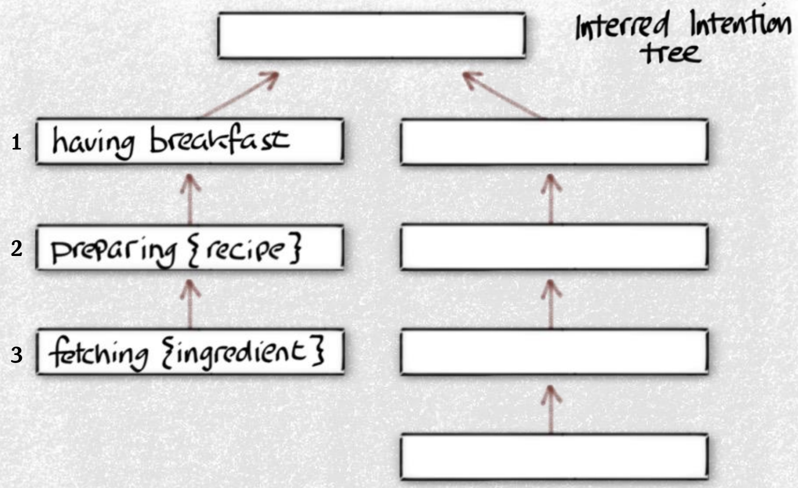


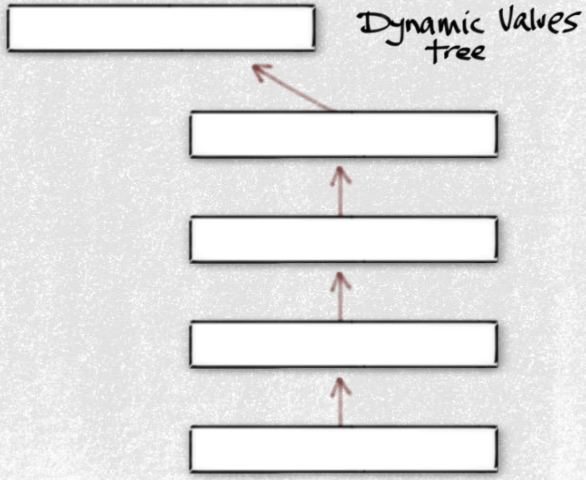


1	*having breakfast*	home.cpp	100
2	*preparing {recipe}*	cooking.cpp	101
3	*fetching {ingredient}*	cooking.cpp	102

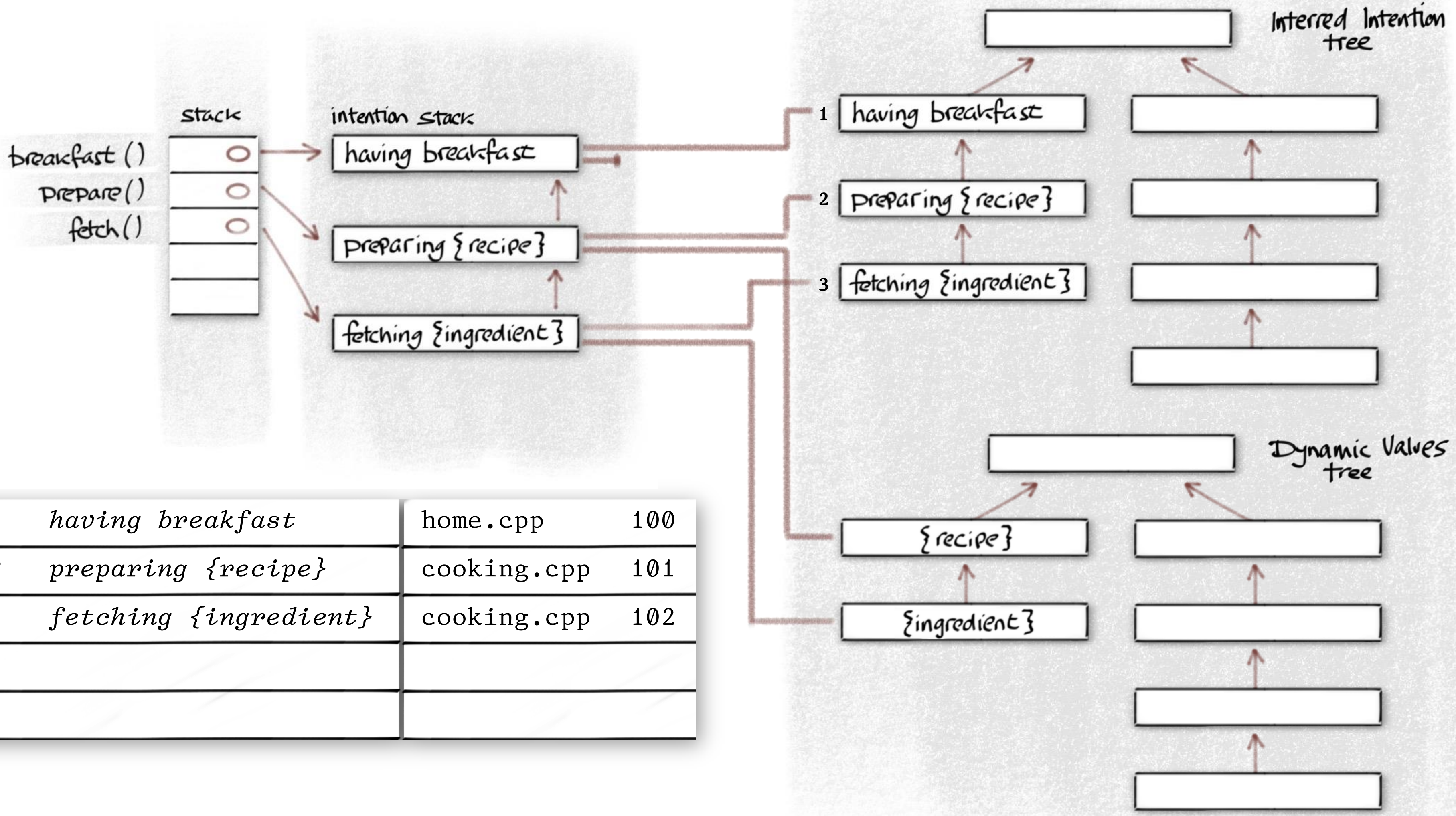


1	*having breakfast*	home.cpp	100
2	*preparing {recipe}*	cooking.cpp	101
3	*fetching {ingredient}*	cooking.cpp	102

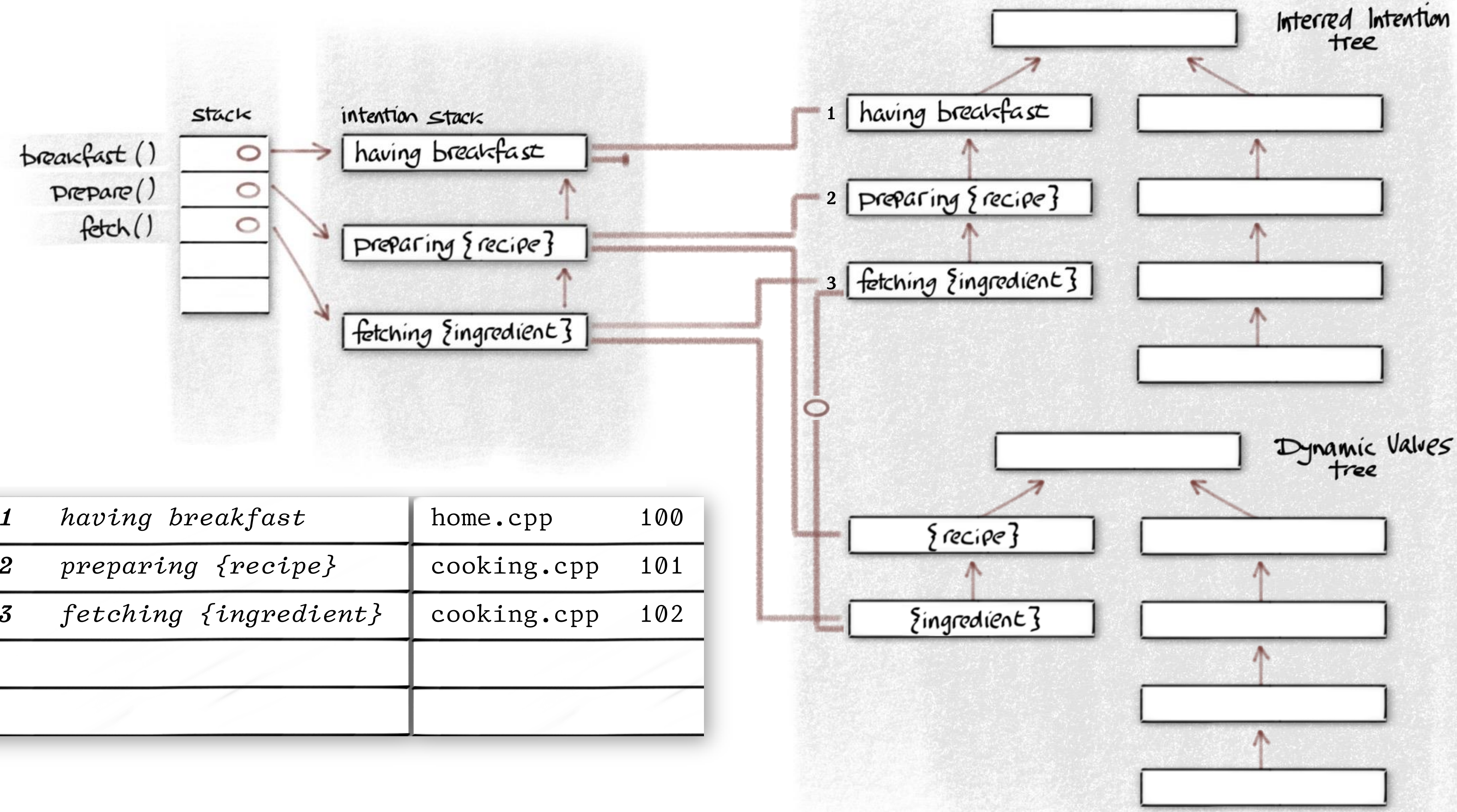




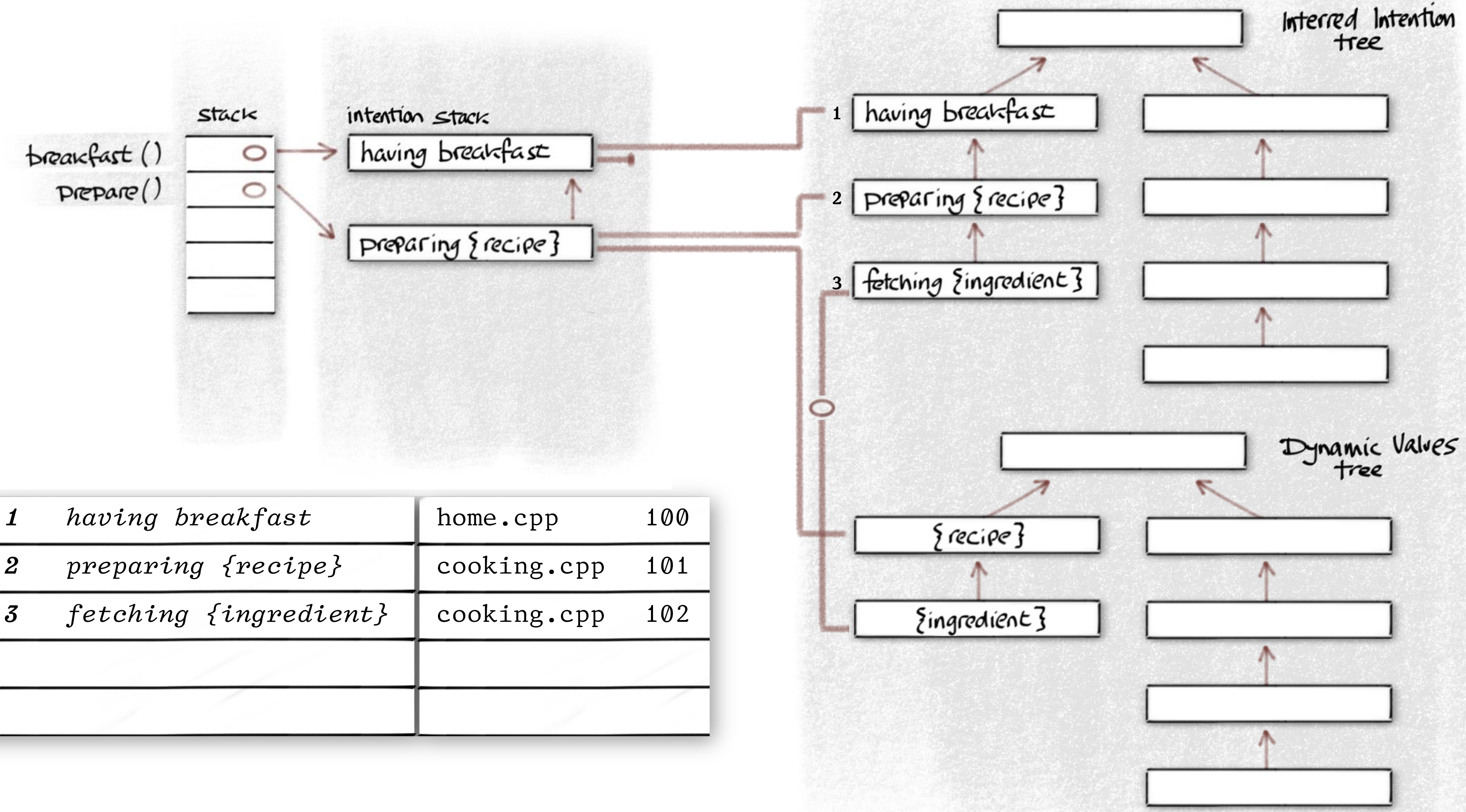
Capture



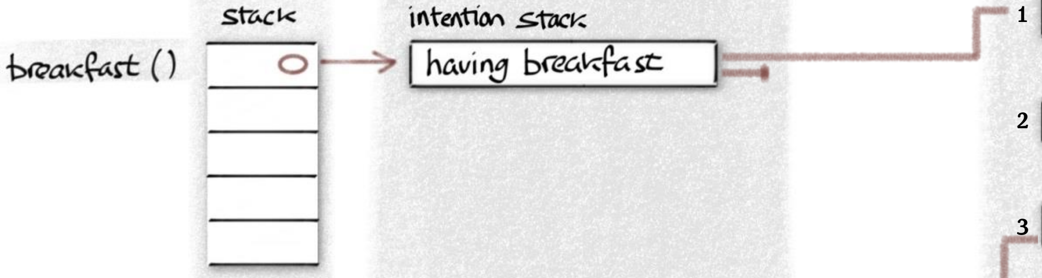
1	*having breakfast*	home.cpp	100
2	*preparing {recipe}*	cooking.cpp	101
3	*fetching {ingredient}*	cooking.cpp	102

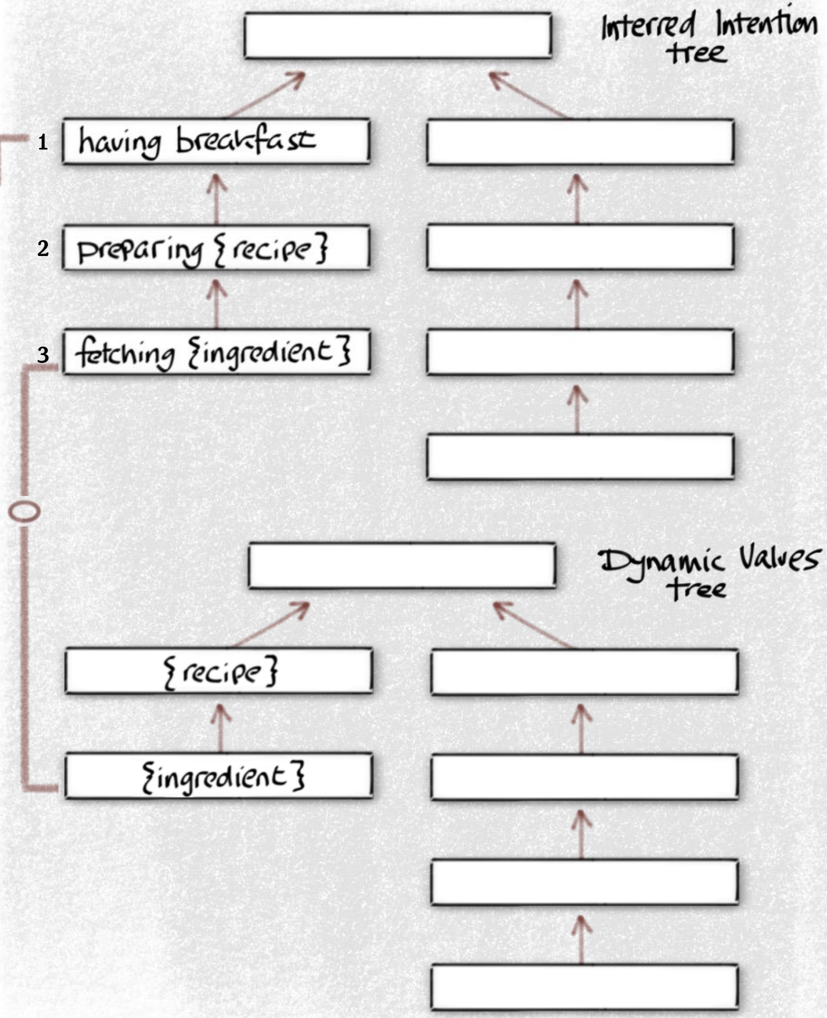


1	*having breakfast*	home.cpp	100
2	*preparing {recipe}*	cooking.cpp	101
3	*fetching {ingredient}*	cooking.cpp	102

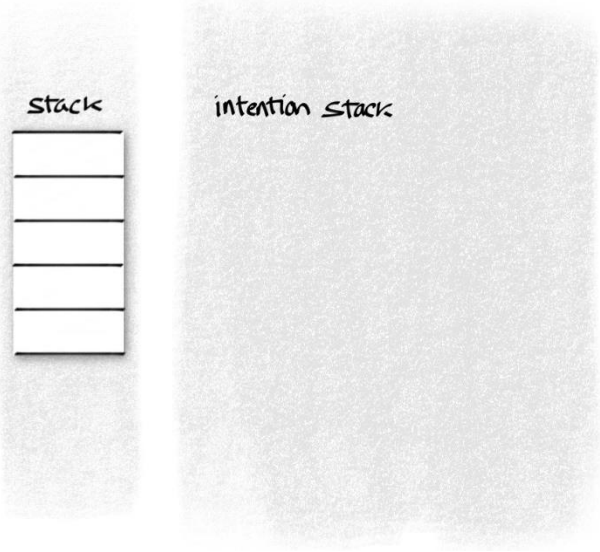


1	*having breakfast*	home.cpp	100
2	*preparing {recipe}*	cooking.cpp	101
3	*fetching {ingredient}*	cooking.cpp	102

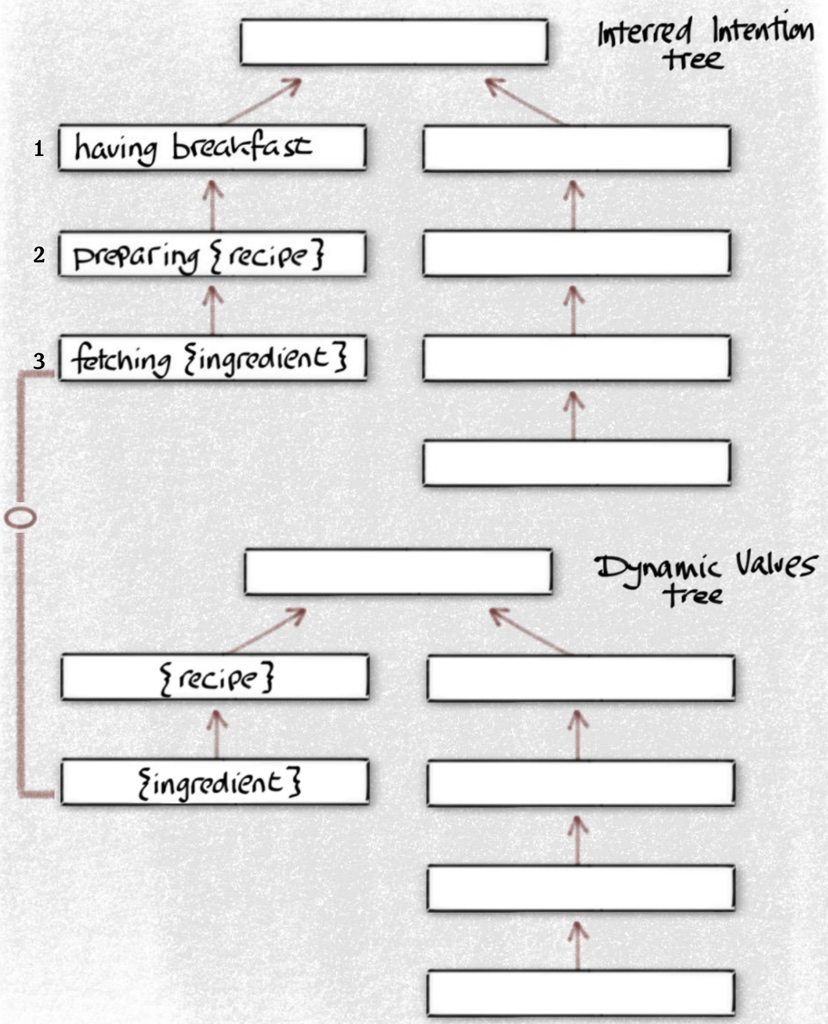




1	*having breakfast*	home.cpp	100
2	*preparing {recipe}*	cooking.cpp	101
3	*fetching {ingredient}*	cooking.cpp	102



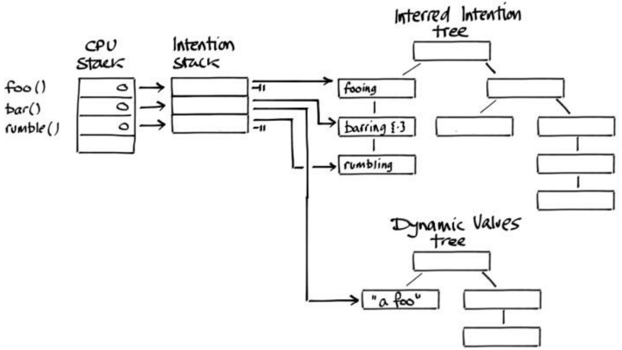
1	*having breakfast*	home.cpp	100
2	*preparing {recipe}*	cooking.cpp	101
3	*fetching {ingredient}*	cooking.cpp	102



Efficiency

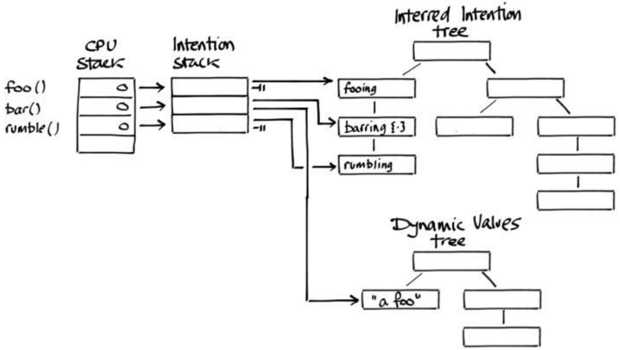
Efficiency

- Intention values (if used) are added to an immutable value tree that may be shared after intention capture. Nodes are reference counted and deleted when no longer required.



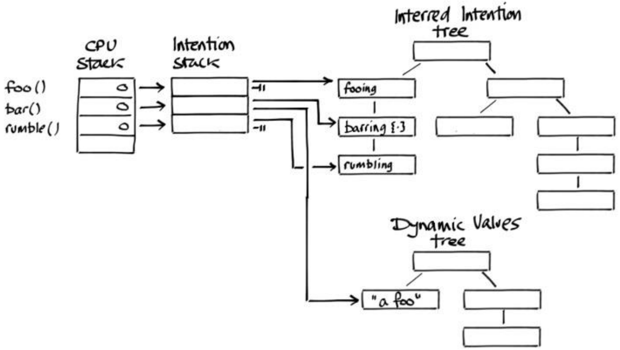
Efficiency

- Intention values (if used) are added to an immutable value tree that may be shared after intention capture. Nodes are reference counted and deleted when no longer required.



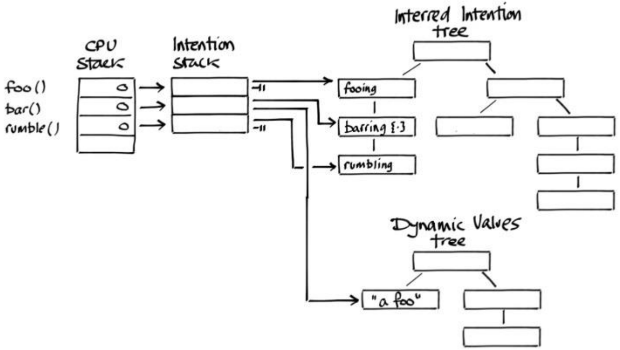
Efficiency

- Overheads are only incurred when intention frames are used.

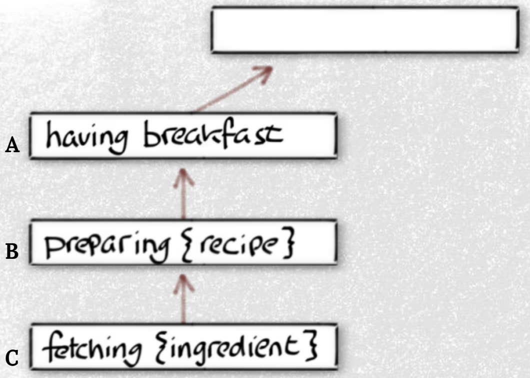


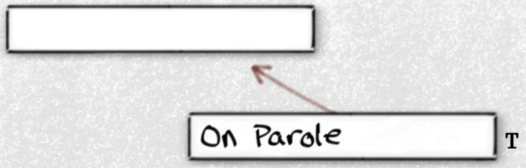
Efficiency in a distributed system

- The interred intention tree can be replicated incrementally when intentions are serialised into messages sent between processes or hosts.



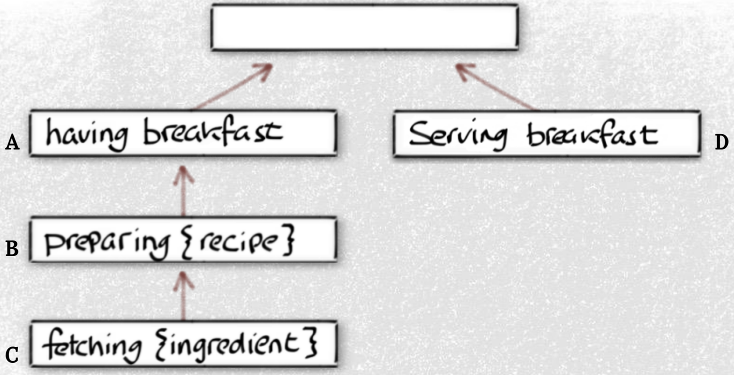
Replication

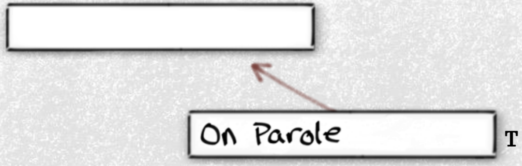




1	*having breakfast*
2	*preparing {recipe}*
3	*fetching {ingredient}*

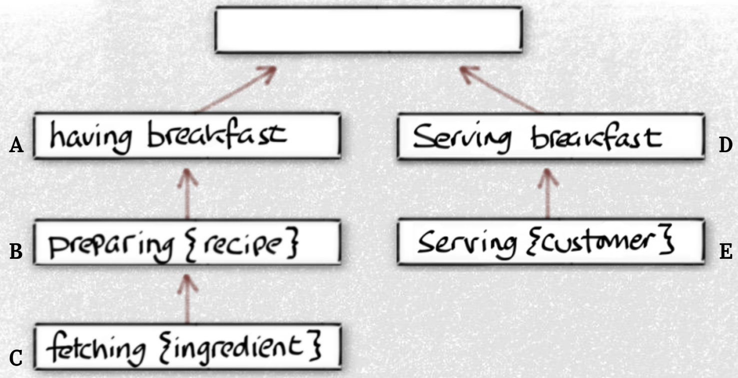
on parole	1

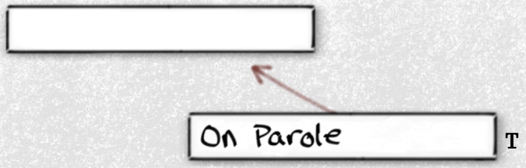




1	*having breakfast*
2	*preparing {recipe}*
3	*fetching {ingredient}*
4	*serving breakfast*

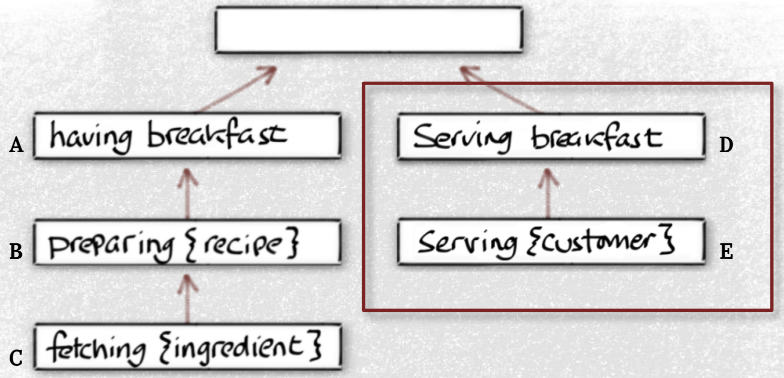
on parole	1

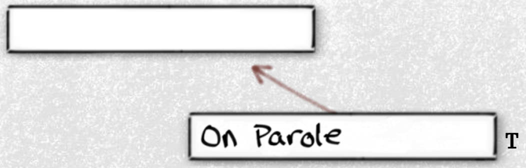




1	*having breakfast*
2	*preparing {recipe}*
3	*fetching {ingredient}*
4	*serving breakfast*
5	*serving {customer}*

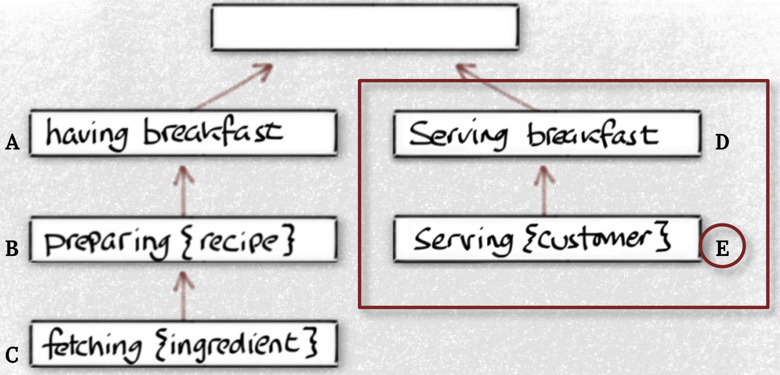
on parole	1



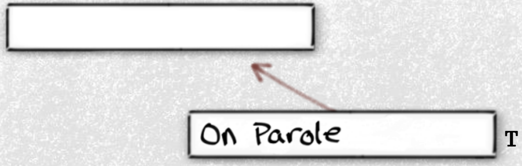


1	*having breakfast*
2	*preparing {recipe}*
3	*fetching {ingredient}*
4	*serving breakfast*
5	*serving {customer}*

on parole	1

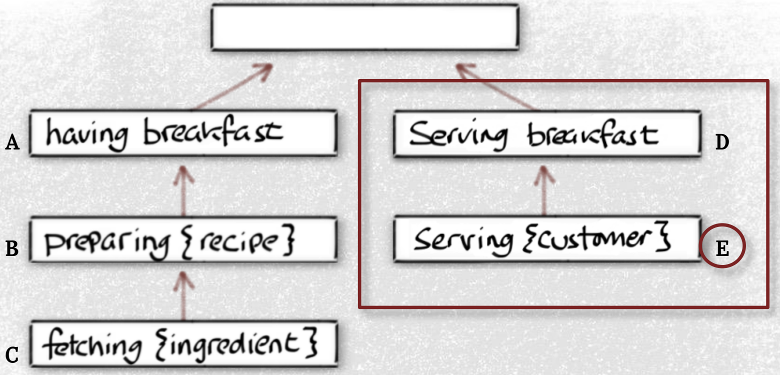


4, 5, E(#4,#5) →

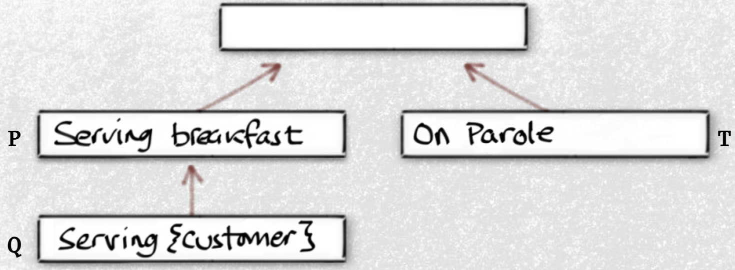


1	*having breakfast*
2	*preparing {recipe}*
3	*fetching {ingredient}*
4	*serving breakfast*
5	*serving {customer}*

on parole	1



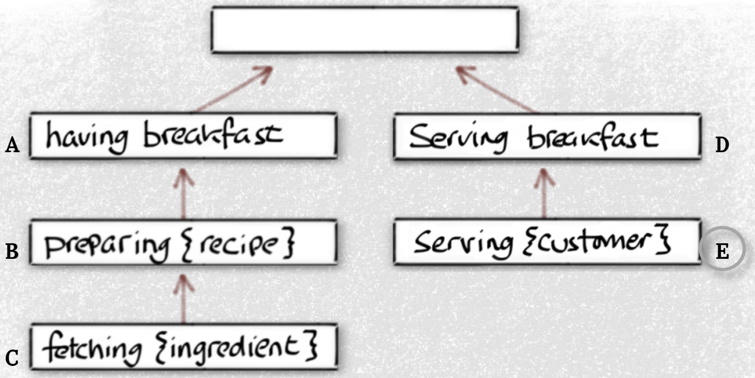
4, 5, E(#4,#5) →

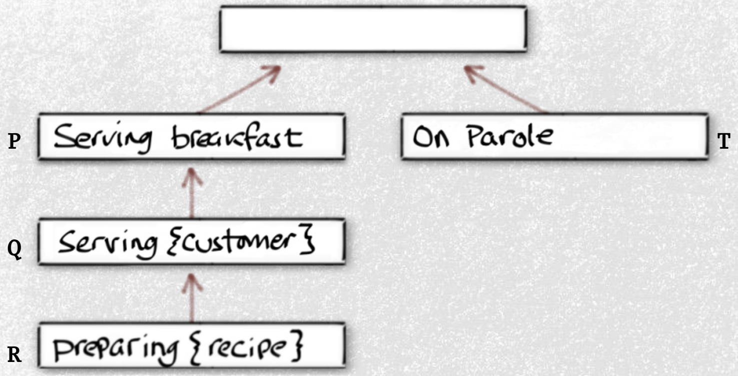


1	*having breakfast*
2	*preparing {recipe}*
3	*fetching {ingredient}*
4	*serving breakfast*
5	*serving {customer}*

	on parole	1
4 → 2	***serving breakfast***	2
5 → 3	***serving {customer}***	3

E → Q

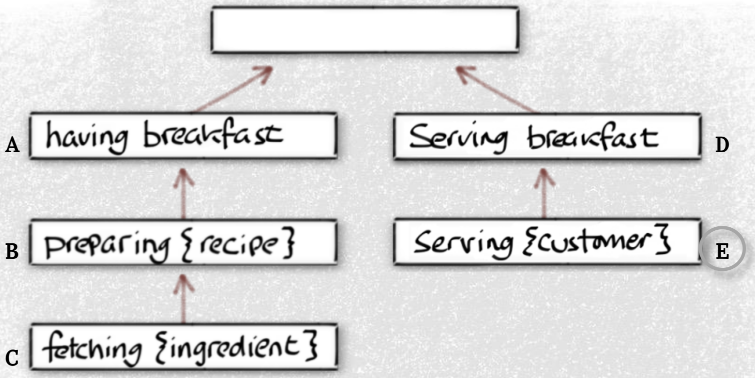


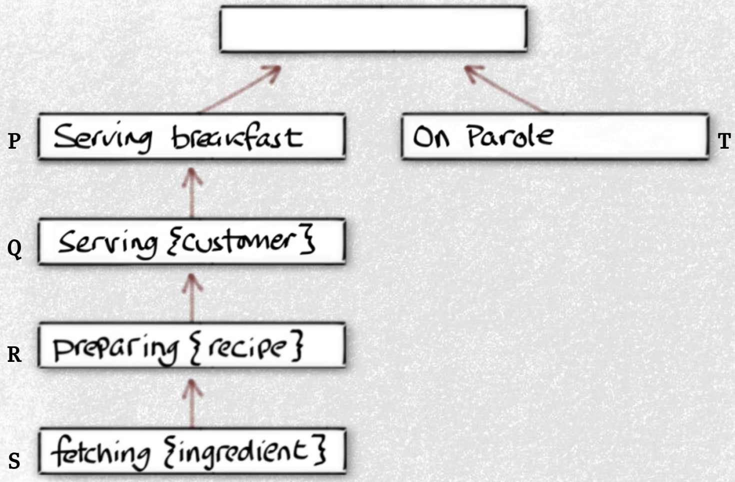


1	*having breakfast*
2	*preparing {recipe}*
3	*fetching {ingredient}*
4	*serving breakfast*
5	*serving {customer}*

	on parole	1
4 → 2	*serving breakfast*	2
5 → 3	*serving {customer}*	3
	preparing {recipe}	4

E → Q

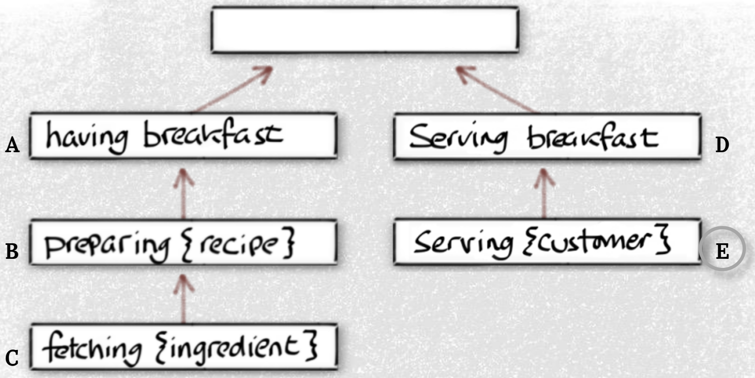


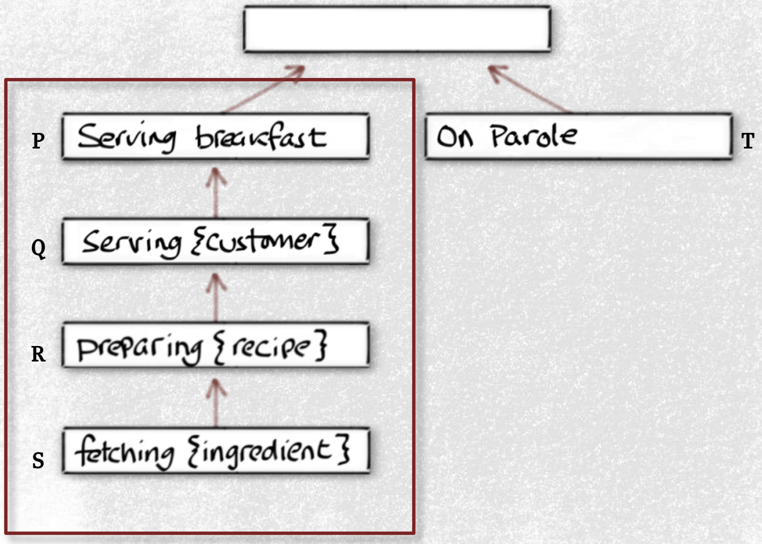


1	*having breakfast*
2	*preparing {recipe}*
3	*fetching {ingredient}*
4	*serving breakfast*
5	*serving {customer}*

	on parole	1
4 → 2	*serving breakfast*	2
5 → 3	*serving {customer}*	3
	preparing {recipe}	4
	fetching {ingredient}	5

E → Q



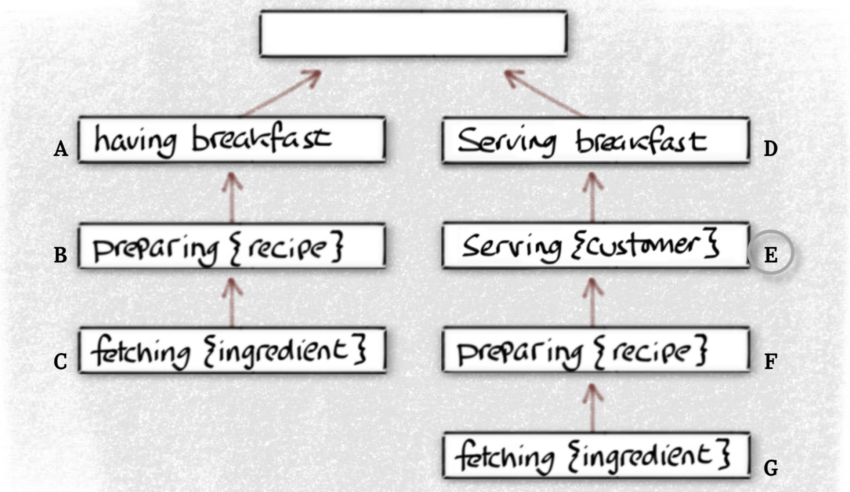


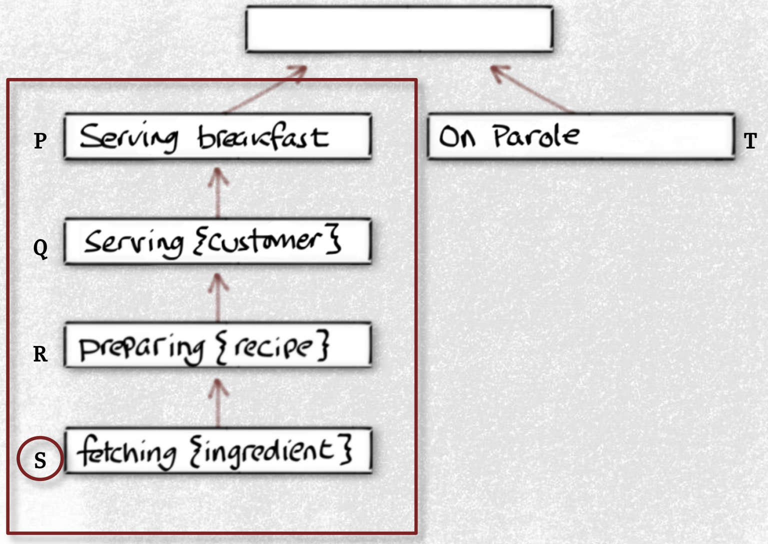
← 2, 3, 4, 5, S(#2,#3,#4,#5)

1	*having breakfast*
2	*preparing {recipe}*
3	*fetching {ingredient}*
4	*serving breakfast*
5	*serving {customer}*

	on parole	1
4 → 2	*serving breakfast*	2
5 → 3	*serving {customer}*	3
	preparing {recipe}	4
	fetching {ingredient}	5

E → Q





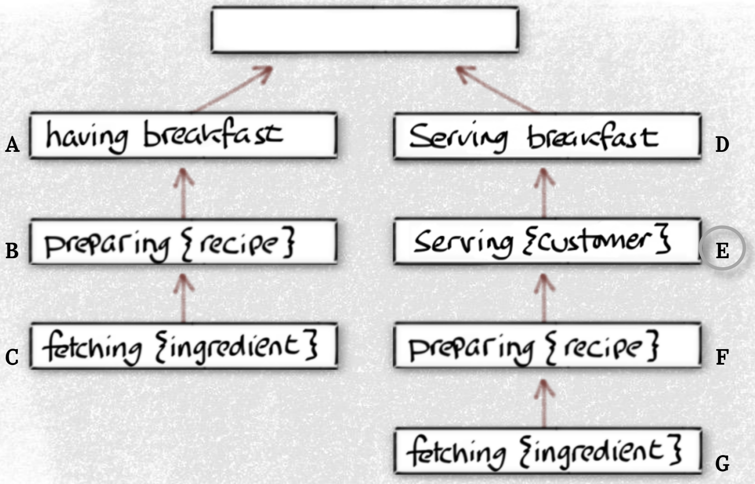
← 2, 3, 4, 5, S(#2,#3,#4,#5)

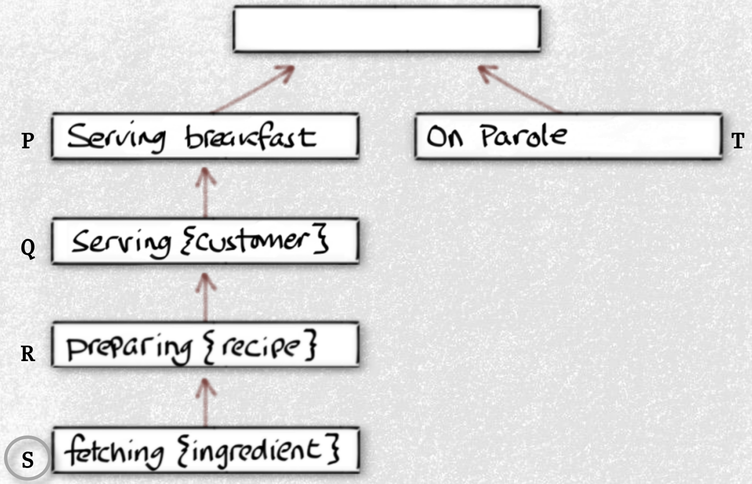
1	*having breakfast*
2	*preparing {recipe}*
3	*fetching {ingredient}*
4	*serving breakfast*
5	*serving {customer}*

2 ← 4
3 ← 5
4 ← 2
5 ← 3
G ← S

4 → 2
5 → 3
E → Q

on parole	1
serving breakfast	2
serving {customer}	3
preparing {recipe}	4
fetching {ingredient}	5



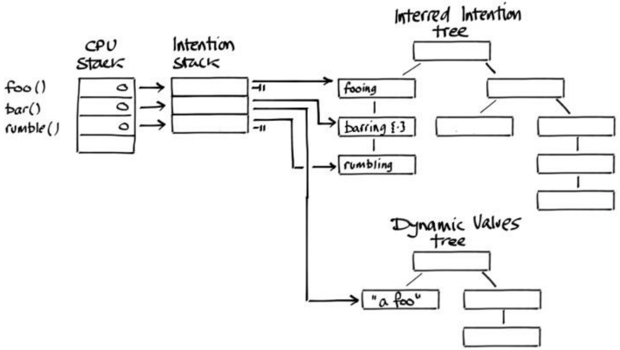


1	*having breakfast*	
2	*preparing {recipe}*	*2* ← 4
3	*fetching {ingredient}*	*3* ← 5
4	*serving breakfast*	*4* ← 2
5	*serving {customer}*	*5* ← 3
		G ← S

	on parole	1
4 → 2	*serving breakfast*	2
5 → 3	*serving {customer}*	3
	preparing {recipe}	4
	fetching {ingredient}	5
E → Q		

Efficiency in a distributed system

- The representation of an intention is only ever transferred once between any two nodes.
- Values must still be transferred each time (but may themselves share their representation).



- So what else do we get for our money?

And…?

Part IV

Archaeology

Logging

Logging

- Intention frames and exceptions can be logged in a compact form.

```
   #1  having breakfast (home.cpp : 100)
   →1
   #2  preparing {recipe} (cooking.cpp : 101)
   →2  “bacon and eggs”
   #3  fetching {ingredient} (cooking.cpp : 102)
   →3  “bacon”
    ←
   →3  “eggs”
    !
    !
    !
    e  the cupboard was bare
    ←
   #4  serving breakfast (cafe.cpp 103)
   →4
   #5  serving {customer} (cafe.cpp 104)
   →5  “dominic”
+λ[n]
    ←
   #6  on parole (kitchen.cpp 100)
   →6
→λ[n]
   →2  “bacon and eggs”
   →3  “bacon”
    ←
   →3  “eggs”
    ←
    ←
```

Hypothetical Format

breakfast

```
 #1  having breakfast (home.cpp : 100)
 →1
 #2  preparing {recipe} (cooking.cpp : 101)
 →2  "bacon and eggs"
 #3  fetching {ingredient} (cooking.cpp : 102)
 →3  "bacon"
  ←
 →3  "eggs"
  !
  !
  !
  e  the cupboard was bare
  ←
 #4  serving breakfast (cafe.cpp 103)
 →4
 #5  serving {customer} (cafe.cpp 104)
 →5  "dominic"
+λ[n]
  ←
 #6  on parole (kitchen.cpp 100)
 →6
→λ[n]
 →2  "bacon and eggs"
 →3  "bacon"
  ←
 →3  "eggs"
  ←
  ←
```

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

breakfast

```
#1  having breakfast (home.cpp : 100)
→1
#2  preparing {recipe} (cooking.cpp : 101)
→2  "bacon and eggs"
#3  fetching {ingredient} (cooking.cpp : 102)
→3  "bacon"
 ←
→3  "eggs"
 !
 !
 !
 e  the cupboard was bare
 ←
#4  serving breakfast (cafe.cpp 103)
→4
#5  serving {customer} (cafe.cpp 104)
→5  "dominic"
+λ[n]
 ←
#6  on parole (kitchen.cpp 100)
→6
→λ[n]
→2  "bacon and eggs"
→3  "bacon"
 ←
→3  "eggs"
 ←
 ←
```

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  prepare(fav);
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

```
#1  having breakfast (home.cpp : 100)
→1
#2  preparing {recipe} (cooking.cpp : 101)
→2  "bacon and eggs"
#3  fetching {ingredient} (cooking.cpp : 102)
→3  "bacon"
 ←
→3  "eggs"
 !
 !
 !
 e  the cupboard was bare
 ←
#4  serving breakfast (cafe.cpp 103)
→4
#5  serving {customer} (cafe.cpp 104)
→5  "dominic"
→λ[n]
 ←
#6  on parole (kitchen.cpp 100)
→6
→λ[n]
→2  "bacon and eggs"
→3  "bacon"
 ←
→3  "eggs"
 ←
 ←
```

the cafe

```
void breakfast_service() {
  whilst("serving breakfast");
  while (customers.waiting())
    take_order(customers.dequeue());
  }
}

void take_order(customer c) {
  whilst("serving {customer}", c);
  orders.queue(order(c,
                     c.choice(),
                     current_intentions()));
}
```

→4 →5
"dominic"

the kitchen

```
#1  having breakfast (home.cpp : 100)
→1
#2  preparing {recipe} (cooking.cpp : 101)
→2  "bacon and eggs"
#3  fetching {ingredient} (cooking.cpp : 102)
→3  "bacon"
 ←
→3  "eggs"
 !
 !
 !
 e  the cupboard was bare
 ←
#4  serving breakfast (cafe.cpp 103)
→4
#5  serving {customer} (cafe.cpp 104)
→5  "dominic"
+λ[n]
 ←
#6  on parole (kitchen.cpp 100)
→6
→λ[n]
→2  "bacon and eggs"
→3  "bacon"
 ←
→3  "eggs"
 ←
 ←
```

```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```

the kitchen

```
#2  preparing {recipe} (cooking.cpp : 101)
→2  "bacon and eggs"
#3  fetching {ingredient} (cooking.cpp : 102)
→3  "bacon"
 ←
→3  "eggs"
 !
 !
 !
 e  the cupboard was bare
 ←
#4  serving breakfast (cafe.cpp 103)
→4
#5  serving {customer} (cafe.cpp 104)
→5  "dominic"
+λ[n]
 ←
#6  on parole (kitchen.cpp 100)
→6
→λ[n]
→2  "bacon and eggs"
→3  "bacon"
 ←
→3  "eggs"
 ←
 ←
 ←
-λ[n]
```

```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```

```
#2  preparing {recipe} (cooking.cpp : 101)
→2  "bacon and eggs"
#3  fetching {ingredient} (cooking.cpp : 102)
→3  "bacon"
 ←
→3  "eggs"
 !
 !
 !
 e  the cupboard was bare
 ←
#4  serving breakfast (cafe.cpp 103)
→4
#5  serving {customer} (cafe.cpp 104)
→5  "dominic"
+λ[n]
 ←
#6  on parole (kitchen.cpp 100)
→6
→λ[n]
→2  "bacon and eggs"
→3  "bacon"
 ←
→3  "eggs"
 ←
 ←
 ←
-λ[n]
```

the kitchen

```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                           std::current_exception(),
                           current_intentions()));
  }
}
```

→4 →5
"dominic"

the kitchen

```
#2  preparing {recipe} (cooking.cpp : 101)
→2  "bacon and eggs"
#3  fetching {ingredient} (cooking.cpp : 102)
→3  "bacon"
 ←
→3  "eggs"
 !
 !
 !
 e  the cupboard was bare
 ←
#4  serving breakfast (cafe.cpp 103)
→4
#5  serving {customer} (cafe.cpp 104)
→5  "dominic"
+λ[n]
 ←
#6  on parole (kitchen.cpp 100)
→6
→λ[n]
→2  "bacon and eggs"
→3  "bacon"
 ←
→3  "eggs"
 ←
 ←
 ←
-λ[n]
```

```
void kitchen_worker() {
  whilst("on parole");
  while (orders.waiting()) {
    prepare_order(orders.dequeue());
  }
}

void prepare_order(order o) {
  with_intent(o.intent());
  try {
    prepare(o.recipe());
  } catch(...) {
    problems.queue(problem(o,
                   std::current_exception(),
                   current_intentions()));
  }
}
```

Part V

Agent Provocateur

- But first, a tip…

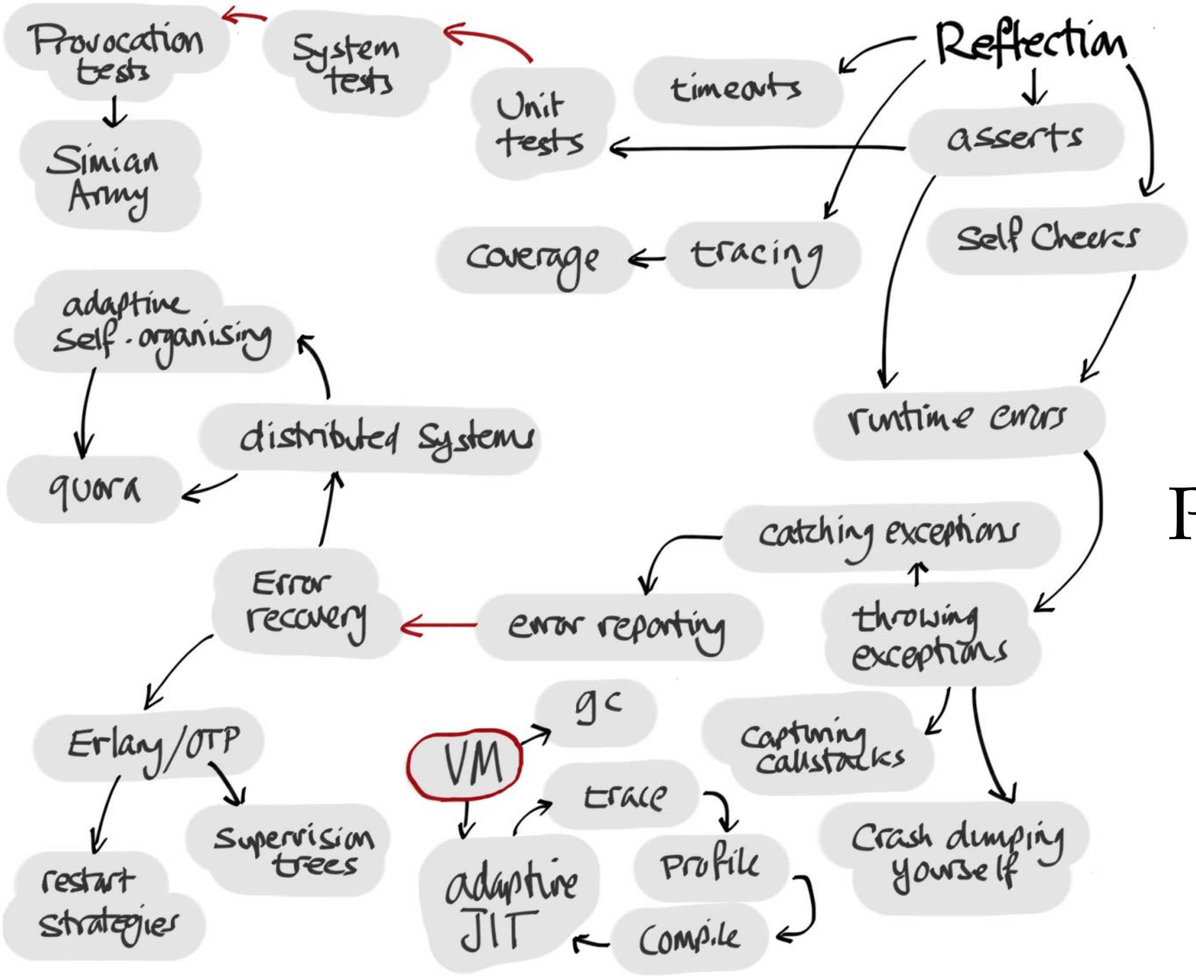
Agent Provocateur

Agent Provocateur

- … don’t Google this at work looking for images to enliven your title slide.

Agent Provocateur

- … don't Google this at work looking for images to enliven your title slide.
- here is one I drew instead…



Agent Provocateur

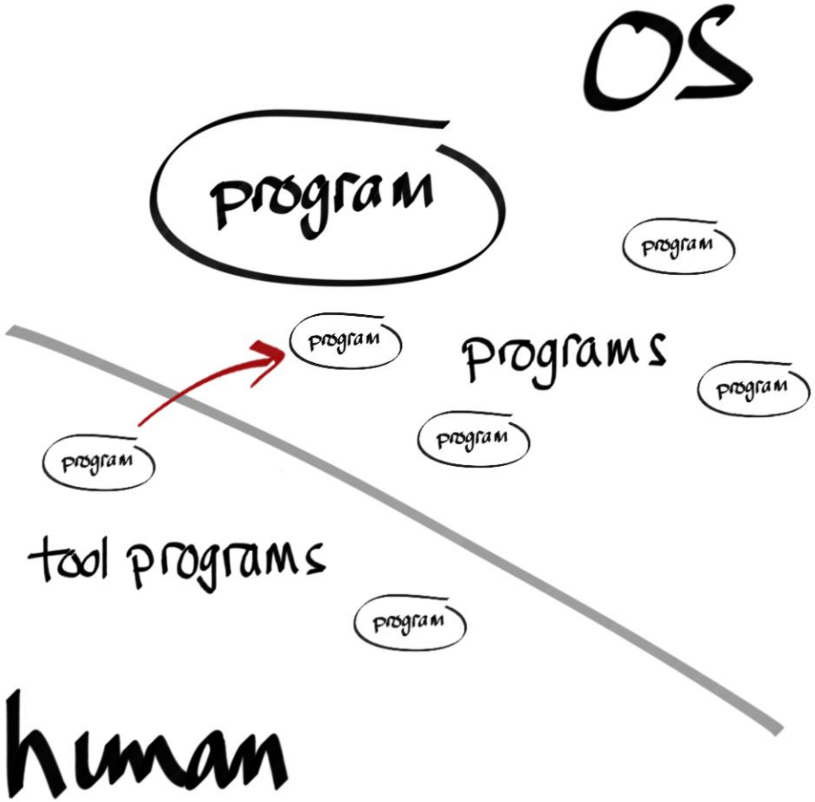
- Intention frames mark scopes in the code where domain relevant activity happens.

- There is an implicit expectation that the activity may fail.

Provocation

Provocation

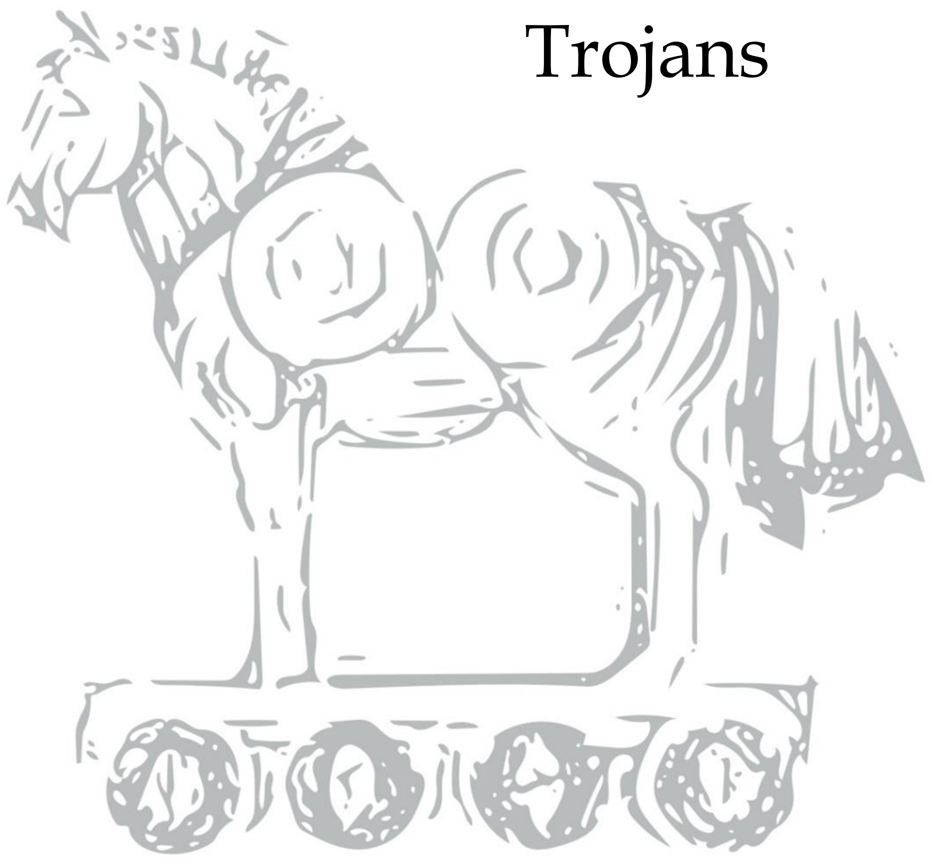
- Intention frames mark scopes in the code where domain relevant activity happens.
- There is an implicit expectation that the activity may fail.
- So… we could test an application's resilience in a controlled way by deliberately provoking errors at these points.



Agent
Provocateur

Trojans

- Inside the horse…



Trojans

- The intention runtime has access to the application as it starts its intended activity.
- It can inspect the application's intentions and selectively inject exceptions to manipulate *effect*.
- It can monitor the application's reaction by observing intention flow in response to it.

Specificity

- By matching specific values in the intention stack, provocations can target and monitor execution flow of specific work items.

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  try {
    whilst("hoping for {favourite}", fav);
    prepare(fav);
  } catch(...) {
    shelve(std::current_exception(),
           current_intentions());
    whilst("making do with {fallback}", toast);
    prepare(toast);
  }
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

“ **whilst** having breakfast

whilst hoping for ***bacon and eggs***
whilst preparing ***bacon and eggs***
whilst fetching ***eggs***
the cupboard was bare

whilst making do with ***toast***
whilst preparing ***toast***
whilst fetching ***bread***
the cupboard was bare ”

```
void breakfast(recipe &fav) {
  whilst("having breakfast");
  try {
    whilst("hoping for {favourite}", fav);
    prepare(fav);
  } catch(...) {
    shelve(std::current_exception(),
           current_intentions());
    whilst("making do with {fallback}", toast);
    prepare(toast);
  }
}

void prepare(recipe &r) {
  whilst("preparing {recipe}", r);
  for(const auto &i : r.ingredients()) {
    fetch(i);
  }
}

void fetch(ingredient &i) {
  whilst("fetching {ingredient}", i);
  cupboard.get(i);
}
```

“ **whilst** having breakfast

whilst hoping for **{FAVOURITE}**
 whilst preparing **{RECIPE}**
  whilst fetching **{INGREDIENT}**
   {EXCEPTION}

whilst making do with **{FALLBACK}**
 whilst preparing **{RECIPE}**
  whilst fetching **{INGREDIENT}**
   {EXCEPTION} ”

Synchronicity

- Trojans can communicate with their controller to coordinate provocation of parallel and distributed systems.
- Waiting until multiple flows have reached specific points by blocking each until conditions are met to release or interrupt them.
- Testing response to:
  - Simultaneous failures.
  - Repeated failures.
  - Induced timeouts.
  - Dropping connections at specific states in a protocol.

- Provided intentions are expressed in terms of domain work rather than implementation details, intention matching patterns used in tests ought to be resilient to implementation change.

Resilience

Harvesting

- Intention descriptions can be harvested statically from source code both to validate patterns used in tests and to generate provocation attack patterns.

- Intention flows can be harvested dynamically via the runtime to collect coverage and to generate context specific provocation patterns.

Command and control

- The intention runtime provides an external command and control interface.
  - Load and unload trojans.
  - Observe intention flow.
  - Coordinate actions at trigger points:
    - Delay.
    - Block until released.
    - Inject exception.

Tests

- Custom test controllers to observe intentions and orchestrate provocations must be succinct and easy to write.
- Use a **declarative** intention matching DSL to target trigger points.
- Employ **actors** and **composable promises** to:
  - Represent and observe triggers.
  - Capture sequences of events.
  - Express expected sequences of events.
  - Hide (some of) the complexities of dealing with asynchronous events.

Caveat

- This doesn’t exist yet…
- … but all the pieces do.

In future

- In a target system implemented with ***intentions*** and ***composable promises*** to reify the forward flow of values, a test system could manipulate both aspects of *effect*:

- *values* and *exceptions*.

Cautions

- Don’t ship builds with the C&C interface.

In conclusion

- *Intentions* are a mechanism for programs to annotate their own execution flow with domain intent.
- They provide a context for exceptions when generating error descriptions.
- They enable a succinct logging mechanism.
- They offer possibilities for program monitoring and provocation testing.

Questions and feedback

- dominic_robinson@sn.scee.net