Pragmatics of Rust and C++: The implementation of a window manager

Max van Deurzen

June 11, 2021

Technische Universität München

Agenda

Agenda

1. What is *Pragmatics*?
2. The *Common Objective*
3. External Dependency Management
4. Main Event Loop
5. Input Bindings
6. Clients
7. Results
8. Discussion

Pragmatics

Definition

1. **Syntax**
   Set of rules that define the *structure* and *composition* of allowable symbols into correct statements or expressions in the language

2. **Semantics**
   The *meaning* of these syntactically valid statements or expressions

3. **Pragmatics**
   "...[T]he third general area of language description, referring to practical aspects of how constructs and features of a language may be used to achieve various objectives."

   Robert D. Cameron, 2002

Example: Assignment Statements

1. **Syntax** (*structure*)

   $x = y * 3;$

2. **Semantics** (*meaning*)
   - x

     Location in memory
   - $y * 3$

     Computation of a value based on an expression
   - $x = y * 3;$

     Store result of expression evaluation in location in memory
3. **Pragmatics** (*purpose*)

   Which objectives are assignment statements used *for*?
   - Setting up a temporary variable used to swap the values of two variables
   - Modifying some part of a compound data structure
   - ...

The Common Objective

Case Study: The implementation of a window manager

- **System Software**
  - Low-level
  - Platform-specific
- **Medium to Large-Sized**
  - Increased Risk of *Code Smells*
    - Monolithic classes
    - Global data
    - High interdependence (Coupling)
    - ...
- **Event-Driven**
  - Reacts to windowing system events
  - Deterministic event dispatch

Case Study: The implementation of a window manager

- **External Dependency Management**
  - Package management
  - Abstracting and decoupling
- **Main Event Loop**
  - Windowing system events
  - Internal events
  - Event dispatch
- **Input Bindings**
  - Storing and retrieving callable objects
- **Clients**
  - Distributed, mutable state

Case Study: The implementation of a window manager

- **Implemented in C++**:
  - wmCPP
- **Implemented in Rust**:
  - wmCPP

External Dependency Management

- The ability to *aid* the programmer in managing external code
  - Automatically downloading a dependency's source code
  - Built-in version control
  - Conflict detection
- Part of the *ecosystem* of a language
  - Installed with its compiler or development environment
- A *must* for any modern programming language

- *No* official package manager
- *Ad hoc* package management
  - Third-party package management tools
  - Custom configure and build scripts
  - Let the user manage the dependencies themselves (e.g. through their distribution's package manager)
- Example: Make script

```
CXXFLAGS = -std=c++17 -march=native -O3
LDFLAGS = `pkg-config --libs x11 xrandr` -flto

obj/%.o: src/%.cc
  g++ ${CXXFLAGS} -MMD -c $< -o $@
all: obj/%.o
  g++ $< ${LDFLAGS} -o bin/wmCPP
```

- *Cargo*, Rust's official package manager
  - Automatically downloads and compiles dependencies
  - A Rust project is a Cargo *package*
  - A package is a collection of *source files* plus a *manifest file*
  - The manifest file describes the package's *meta-information*, *dependencies*, and a set of *target crates*
  - A crate represents a *library* or *binary executable* program
- Example: `Cargo.toml` manifest file

```
[package]
name = "wmRS"
version = "0.1.0"
edition = "2018"
license = "BSD3"
default-run = "wmRS"
description = """
An ICCCM & EWMH compliant X11 reparenting,
tiling window manager, written in Rust
"""
```

Decoupling Dependencies

- What is a *Windowing System*?
  - System software
  - Responsible for providing graphical primitives to construct and present GUIs
  - Render applications' windows' contents
  - Main dependency of the window manager
- The window manager's implementation is made *agnostic* to that of the windowing system
  - We create a *library*, `winsys`, that represents an abstraction above and wrapper around the API into the windowing system
    - The library defines an *interface* that outlines desired behavior
    - The interface is *implemented* for each supported windowing system
  - The connection with windowing system is *decoupled* from the implementation of the window manager

To define an interface that represents the connection with the windowing system, we use a Rust *trait*

- Traits are *zero-overhead* collections of methods that are:
  - Declared for *some* type `Self`
  - Most often used to implement *shared behavior*
- Traits most closely resemble the concept of *interfaces* from other languages
- Traits can provide a *default implementation* for their defined methods
- Example: `wmRS`'s `Connection` trait

```rust
pub trait Connection {
  fn step(&self) -> Option<Event>;
  fn move_window(&self, window: Window, pos: Pos);
  fn resize_window(&self, window: Window, dim: Dim);
  fn close_window(&self, window: Window);
  // ...
}
```

We implement the `Connection` trait to target a *specific* windowing system

- Example: wmRS's `Connection` implementation for the X Window System

```rust
use x11rb::connection;

pub struct XConnection
  <'conn, Conn: connection::Connection>
{
  conn: &'conn Conn,
  // ...
}
impl<'conn, Conn: connection::Connection>
  Connection for XConnection<'conn, Conn>
{
  fn step(&self) -> Option<Event> { /* ... */ }
  // ...
}
```

Main Event Loop

Second Frame

Hello, world!

Input Bindings

Second Frame

Hello, world!

Clients

Second Frame

Hello, world!