Ripgrep haystack.rs: Code Companion

Reference code for the haystack.rs lecture. Sections correspond to the lecture document.

Section 1: The Builder Pattern

Usage in HiArgs:

```
// From hiargs.rs
pub(crate) fn haystack_builder(&self) -> HaystackBuilder {
    let mut builder = HaystackBuilder::new();
    builder.strip_dot_prefix(self.paths.has_implicit_path);
    builder
}
```

Why a builder for one field?

- Consistent API pattern across ripgrep
- Room for future configuration
- Separates construction from usage
- Builder is Clone can be shared across threads

Section 2: Building from Results

How it's used in main.rs

```
// Single-threaded search
let unsorted = args
    .walk_builder()?
    .build()
    .filter_map(|result| haystack_builder.build_from_result(result));

// Parallel search
Box::new(move |result| {
    let haystack = match haystack_builder.build_from_result(result) {
        Some(haystack) => haystack,
        None => return WalkState::Continue, // Skip, keep walking
    };
    // ... search the haystack
})
```

Error handling flow:

Section 3: The Filtering Decision

Decision flowcharts

Section 4: Explicit vs Implicit

```
impl Haystack {
    /// Returns true if and only if this entry corresponds to a haystack to
    /// search that was explicitly supplied by an end user.
    ///
    /// Generally, this corresponds to either stdin or an explicit file path
    /// argument. e.g., in 'rg foo some-file ./some-dir/', 'some-file' is
    /// an explicit haystack, but, e.g., './some-dir/some-other-file' is not.
    ///
    /// However, note that ripgrep does not see through shell globbing. e.g.,
    /// in 'rg foo ./some-dir/*', './some-dir/some-other-file' will be treated
    /// as an explicit haystack.
    pub(crate) fn is_explicit(&self) -> bool {
        // stdin is always explicit
        // depth() == 0 means directly provided, not discovered
        // !is_dir() because directories are never "searched"
        self.is_stdin() || (self.dent.depth() == 0 && !self.is_dir())
    }

    /// Returns true if and only if this entry corresponds to stdin.
    pub(crate) fn is_stdin(&self) -> bool {
        self.dent.is_stdin()
    }
}
```

Examples:

rg foo file.txt	file.txt	
rg foo ./src/	./src/main.rs	
rg foo	<pre>src/main.rs</pre>	
rg foo -		
echo x \ rg foo		
rg foo ./src/*	./src/lib.rs	

Why depth=0 means explicit:

```
// When you run: rg foo file1.txt ./dir/
// The walker sees:
// file1.txt → depth 0 (you provided it)
// ./dir/ → depth 0 (you provided it)
// ./dir/a.rs → depth 1 (walker found it)
// ./dir/sub/b.rs → depth 2 (walker found it)
```

Section 5: Path Display

```
impl Haystack {
    // Return the file path corresponding to this haystack.
    ///
    // If this haystack corresponds to stdin, then a special `<stdin>` path
    // is returned instead.
    pub(crate) fn path(&self) -> &Path {
        if self.strip_dot_prefix && self.dent.path().starts_with("./") {
            self.dent.path().strip_prefix("./").unwrap()
        } else {
            self.dent.path()
        }
    }
}
```

Behavior examples:

Input	strip_dot_prefix	path() returns
./src/main.rs	true	src/main.rs
./src/main.rs	false	./src/main.rs
<pre>src/main.rs</pre>	true	src/main.rs
src/main.rs	false	src/main.rs
		<stdin> (from DirEntry)</stdin>

When strip_dot_prefix is enabled:

```
// In hiargs.rs
builder.strip_dot_prefix(self.paths.has_implicit_path);

// has_implicit_path is true when user ran `rg foo` with no paths
// This makes output cleaner:
// "src/main.rs:42:fn main()"
// Instead of:
// "./src/main.rs:42:fn main()"
```

Section 6: File Type Detection

```
impl Haystack {
    /// Returns true if and only if this haystack points to a directory after
    /// following symbolic links.
    fn is_dir(&self) -> bool {
        let ft = match self.dent.file_type() {
            None => return false, // Can't determine = not a dir
            Some(ft) => ft,
        };
        if ft.is_dir() {
            return true;
        }
        // Symlink that points to a directory?
        self.dent.path_is_symlink() && self.dent.path().is_dir()
    }

    /// Returns true if and only if this haystack points to a file.
    fn is_file(&self) -> bool {
        self.dent.file_type().map_or(false, |ft| ft.is_file())
    }
}
```

File type scenarios:

Filesystem Object			
	true		
		true	
		true	

Why symlink handling differs:

```
// is_dir follows symlinks:
self.dent.path_is_symlink() && self.dent.path().is_dir()

// is_file does NOT follow symlinks:
self.dent.file_type().map_or(false, |ft| ft.is_file())

// Reason: By the time we get here, --follow has already been applied.
// If the user wanted symlinks followed, the walker already did it.
// A symlink that wasn't followed should not be searched.
```

Section 7: The Wrapper Structure

```
/// A haystack is a thing we want to search.
///
/// Generally, a haystack is either a file or stdin.
#[derive(Clone, Debug)]
pub(crate) struct Haystack {
    dent: ignore::DirEntry,
    strip_dot_prefix: bool,
}
```

What ignore::DirEntry provides:

```
// From the ignore crate (not shown in haystack.rs)
impl DirEntry {
    fn path(&self) -> &Path;
    fn file_type(&self) -> Option<FileType>;
    fn metadata(&self) -> Result<Metadata, Error>;
    fn depth(&self) -> usize;
    fn is_stdin(&self) -> bool;
    fn path_is_symlink(&self) -> bool;
    fn error(&self) -> Option<&Error>;
}
```

Why wrap instead of using DirEntry directly?

- 1. Custom path logic strip_dot_prefix behavior
- 2. **Application-level filtering** is_explicit, is_file
- Encapsulation Search code sees Haystack API, not ignore API
- 4 Future flexibility Can add fields without changing ignore crate

Ouick Reference: Full API

Integration Example

Data flow: