

17-695: Design Patterns

Creational Patterns

Wednesday, September 9 • Lecture and Mini-Studio

Administrivia

- A1 grades are now released; regrades are open for one week
- Revisions to planned assignment sequence
 - A4 has been dropped to give more room to group project (no overlap with exam)
 - Final exam will be on **Saturday, October 10th** (time TBD)
 - No more "best three out of four" submissions
 - **You can now revise and resubmit one assignment before the exam**
- Use AI to support you in the homeworks, not to replace you
 - You won't have access to AI in the final exam



By the end of today

Find the decisions

Locate selection, construction, reuse, and cleanup in code.

Choose a mechanism

Use the simplest construction approach that meets the requirement.

Make dependencies explicit

Let callers and tests supply the objects a service needs.

Identify the owner

State who shares each resource and who closes it.

Running Scenario: Launchline



An installable service that **publishes customer builds** from CI to a source-code hosting service.

Used by	Release engineers and CI pipelines
Inputs	repository · version tag · build artifacts
Creates	GitHub release · assets · completed check

```
class ReleaseService:
    def publish(
        self, repo_name: str, tag: str, files: list[str],
    ) -> None:
        token = os.environ["GITHUB_TOKEN"]
        client = GitHubClient(
            token=token,
            base_url="https://api.github.com",
        )
        repository = GitHubRepository(client, repo_name)
        release = GitHubRelease(client, repository.id)

        release.create(tag=tag, draft=False)
        for path in files:
            release.upload(path)

        GitHubChecks(client, repository.id).mark_released(tag)
```

Create the release → upload each file → mark it complete.



A customer needs self-hosted GitLab



NEW DEPLOYMENT REQUIREMENT

- Publish to a GitLab instance on the customer's own network.
-  **The customer wants to see that unit tests pass without contacting GitHub or GitLab.**

What would need to change for GitLab?

```
class ReleaseService:
    def publish(
        self, repo_name: str, tag: str, files: list[str],
    ) -> None:
        token = os.environ["GITHUB_TOKEN"]
        client = GitHubClient(
            token=token,
            base_url="https://api.github.com",
        )
        repository = GitHubRepository(client, repo_name)
        release = GitHubRelease(client, repository.id)

        release.create(tag=tag, draft=False)
        for path in files:
            release.upload(path)

        GitHubChecks(client, repository.id).mark_released(tag)
```

This class is coupled to GitHub: credentials, URL, classes, and API calls.

Coupling: what does a component depend on?

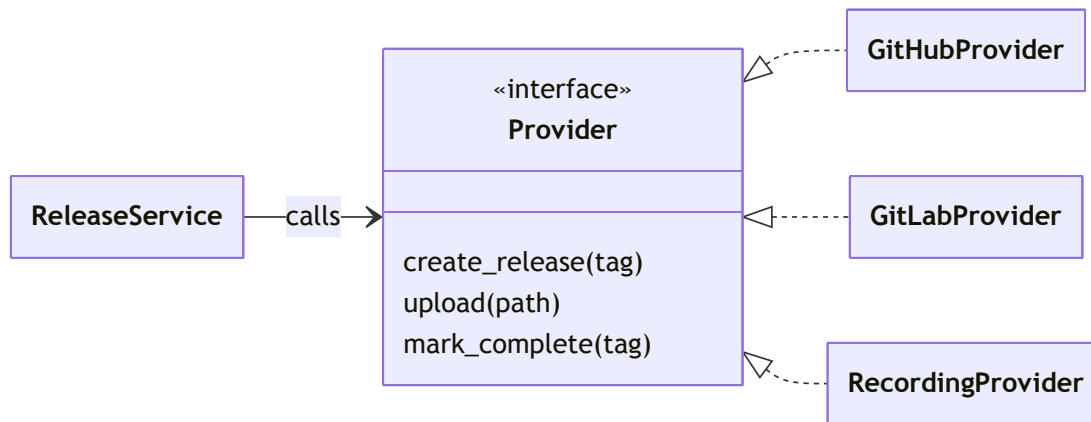


KEY CONCEPT Coupling

The degree of dependence between components: what one must know or assume about another to do its work.

In Launchline: changing hosting services forces changes in `publish()` because it knows GitHub's classes and API.

Decouple the service with a provider interface



**The service knows provider operations;
each provider knows its vendor API.**

Extend behavior through a stable interface



KEY CONCEPT

Open-Closed Principle

Design components to allow new behavior through extension without modifying their existing code.

In Launchline: add a provider that satisfies the contract without rewriting `publish()`.

Which lines configure the objects?

```
class ReleaseService:
    def publish(
        self, repo_name: str, tag: str, files: list[str],
    ) -> None:
        token = os.environ["GITHUB_TOKEN"]
        client = GitHubClient(
            token=token,
            base_url="https://api.github.com",
        )
        repository = GitHubRepository(client, repo_name)
        release = GitHubRelease(client, repository.id)

        release.create(tag=tag, draft=False)
        for path in files:
            release.upload(path)

        GitHubChecks(client, repository.id).mark_released(tag)
```

Construction and configuration sit inside the workflow.

Keep each responsibility together



KEY CONCEPT
Cohesion

The degree to which a component's parts contribute to a focused responsibility.

Release service: controls the sequence: create → upload → complete.

Provider: translates those operations into one vendor's API calls.

Two different reasons to change



KEY CONCEPT

**Single
Responsibility
Principle**

A component should have one reason to change. Separate code that changes for different reasons.

Publishing rule: verify before completion → change the service's sequence.

Provider API: new authentication method → change that provider's implementation.

**The customer wants unit tests
that run without a network.**

Test the workflow with a substitute

Dependency

An object a component needs to do its work, such as the GitHub client.

Test double

A substitute for a real dependency during a test. Ours will record calls without contacting a server.

Testability

How easily we can control the conditions of a test and observe the results.

The method creates its own client

```
class ReleaseService:
    def publish(
        self, repo_name: str, tag: str, files: list[str],
    ) -> None:
        token = os.environ["GITHUB_TOKEN"]
        client = GitHubClient(
            token=token,
            base_url="https://api.github.com",
        )
        repository = GitHubRepository(client, repo_name)
        release = GitHubRelease(client, repository.id)

        release.create(tag=tag, draft=False)
        for path in files:
            release.upload(path)

        GitHubChecks(client, repository.id).mark_released(tag)
```

Poor testability: callers cannot supply test doubles.

Let the caller supply the provider



KEY CONCEPT

Dependency Injection

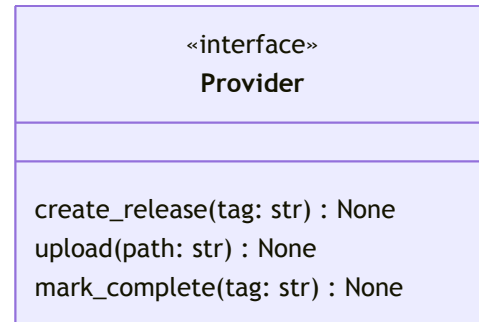
Provide a component's dependencies from outside the component.

In Launchline: the `ReleaseService` constructor accepts a provider. The application supplies a vendor provider; a test supplies a `RecordingProvider` and checks its calls.

What must each provider support?

```
class Provider(Protocol):  
    def create_release(self, tag: str) -> None: ...  
    def upload(self, path: str) -> None: ...  
    def mark_complete(self, tag: str) -> None: ...
```

Each provider is configured for one repository.



Python note: A class satisfies `Provider` if it implements all three methods with compatible signatures. It may have additional methods; inheriting from `Provider` is not required.

We inject the Provider into the ReleaseService

```
class ReleaseService:
    def __init__(self, provider: Provider) -> None:
        self._provider = provider

    def publish(self, tag: str, files: list[str]) -> None:
        self._provider.create_release(tag)
        for path in files:
            self._provider.upload(path)
        self._provider.mark_complete(tag)
```

- The repository name now goes to the provider constructor.
- The tag and files still go to `publish()`.

```
class GitHubProvider:
    def __init__(self, http, api_url, token, repo_name):
        self.client = GitHubClient(
            http=http, token=token, base_url=api_url,
        )
        self.repo_name = repo_name
        repository = GitHubRepository(self.client, self.repo_name)
        self.release = GitHubRelease(self.client, repository.id)
        self.checks = GitHubChecks(self.client, repository.id)

    def create_release(self, tag):
        self.release.create(tag=tag, draft=False)

    def upload(self, path):
        self.release.upload(path)

    def mark_complete(self, tag):
        self.checks.mark_released(tag)
```

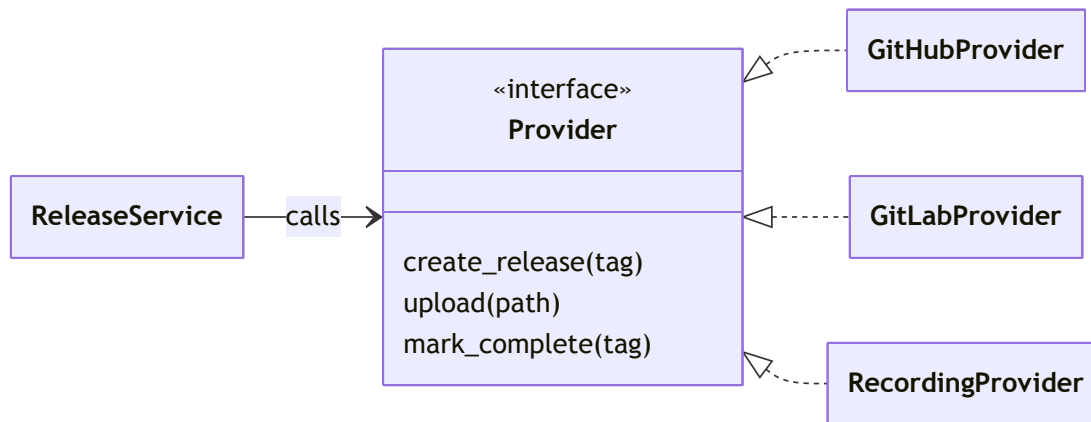
```
class GitHubProvider:
    def __init__(self, http, api_url, token, repo_name):
        self.client = GitHubClient(
            http=http, token=token, base_url=api_url,
        )
        self.repo_name = repo_name
        repository = GitHubRepository(self.client, self.repo_name)
        self.release = GitHubRelease(self.client, repository.id)
        self.checks = GitHubChecks(self.client, repository.id)

    def create_release(self, tag):
        self.release.create(tag=tag, draft=False)

    def upload(self, path):
        self.release.upload(path)

    def mark_complete(self, tag):
        self.checks.mark_released(tag)
```

The same service can use different providers



```
ReleaseService(github_provider).publish(tag, files)
ReleaseService(gitlab_provider).publish(tag, files)
ReleaseService(fake_provider).publish(tag, files)
```

Polymorphism: the service calls the same methods on different implementations.

A fake can record the publishing steps

```
class RecordingProvider:
    def __init__(self):
        self.calls = []

    def create_release(self, tag):
        self.calls.append(("create", tag))

    def upload(self, path):
        self.calls.append(("upload", path))

    def mark_complete(self, tag):
        self.calls.append(("complete", tag))
```

- These methods record calls without contacting a server
- We can use this **Fake** implementation for testing
- We will look at test doubles in much more depth in **Quality Assurance**

We can check the service performs the expected calls without the network

```
provider = RecordingProvider()
service = ReleaseService(provider)

service.publish("2.1.0", ["app.zip"])

assert provider.calls == [
    ("create", "2.1.0"),
    ("upload", "app.zip"),
    ("complete", "2.1.0"),
]
```

Where should the program choose the provider?

The service can use either implementation.

The program must still read the settings and create a GitHub or GitLab provider.

Creational Patterns

Creational patterns: how objects are created

Pattern	Creation decision
Factory Method	Let a subclass choose the product.
Abstract Factory	Create a compatible product family.
Builder	Assemble an object in steps.
Prototype	Copy an existing object.
Singleton	One instance with global access.

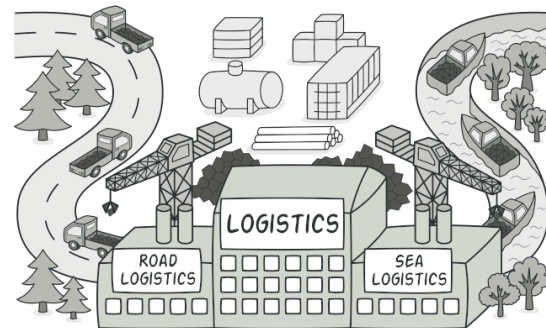


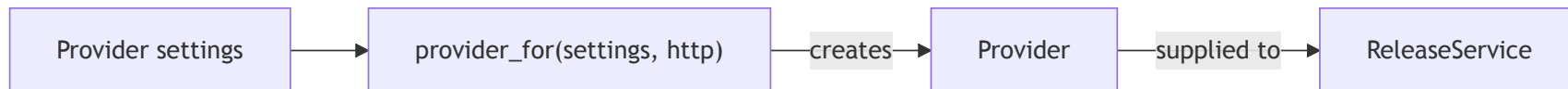
Illustration: [Refactoring.Guru](https://refactoring.guru/) / Dmitry Zhart

We start with a **simple factory function**: select a provider from settings.

Factory Function: Choose a Provider

```
def provider_for(config: ProviderConfig, http: HttpClient) -> Provider:  
    if config.kind == "github":  
        return GitHubProvider(  
            http=http, api_url=config.url,  
            token=config.token, repo_name=config.repository,  
        )  
    if config.kind == "gitlab":  
        return GitLabProvider(  
            http=http, base_url=config.url,  
            access_token=config.token, repo_name=config.repository,  
        )  
    raise UnknownProvider(config.kind)
```

Provider is chosen and configured at startup



```
with HttpClient() as http:  
    provider = provider_for(settings.provider, http)  
    service = ReleaseService(provider)  
    service.publish(tag, files)
```

Python's with block manages resource lifetime

Entering the context binds the client to http .

Leaving the block closes it, even if publishing raises an exception.

New requirement: independently installed providers

A separate team ships **Bitbucket support** as an installable package.

Today, adding it means editing `provider_for` and releasing the core application.



Bitbucket

How can the package supply its own construction code?

1. Give every factory the same `(config, http)` signature.
2. Register each factory under a provider name.
3. Load extensions at startup, then select a provider.

Each provider translates its own configuration

```
class GitHubProvider:
    ...
    @classmethod
    def from_config(cls, config, http):
        return cls(
            http=http, api_url=config.url,
            token=config.token, repo_name=config.repository,
        )

provider = GitHubProvider.from_config(config, http)
```

What does Python supply to each method?

Method	Python supplies	Typical use and call
Instance	The instance: <code>self</code>	Use object state: <code>provider.publish(tag, files)</code>
<code>@classmethod</code>	The class: <code>cls</code>	Alternate constructor: <code>GitHubProvider.from_config(config, http)</code>
<code>@staticmethod</code>	No implicit argument	Helper grouped with a class: <code>GitHubProvider.valid_kind(kind)</code>

What the registry stores

In `GitHubProvider.from_config`, the class is already bound.
Callers supply only `(config, http)`.

We use a registry to store functions by provider name

```
FACTORIES = {  
    "github": GitHubProvider.from_config,  
    "gitlab": GitLabProvider.from_config,  
}  
  
def register_provider(kind, factory):  
    if kind in FACTORIES:  
        raise ValueError(f"duplicate provider: {kind}")  
    FACTORIES[kind] = factory
```

We lookup the registry at construction time to find the appropriate factory function

```
def provider_for(config, http) -> Provider:
    try:
        factory = FACTORIES[config.kind]
    except KeyError:
        raise UnknownProvider(config.kind) from None
    return factory(config, http)
```

- Each registered function has the same interface: It translates those shared arguments into its own implementation's constructor arguments (may be different).

Discovery can automatically register factory functions

The Bitbucket package declares a factory in `pyproject.toml`:

```
[project.entry-points."release_service.providers"]  
bitbucket = "bitbucket_provider:BitbucketProvider.from_config"
```

The application discovers installed factories at startup:

```
from importlib.metadata import entry_points  
  
def load_provider_plugins() -> None:  
    for entry in entry_points(group="release_service.providers"):  
        register_provider(entry.name, entry.load())
```

Discovery registers `"bitbucket"`; a later `provider_for(config, http)` call constructs it when `config.kind == "bitbucket"`.

When does a registry help?

Keep the conditional

- A small, fixed provider set
- One team owns selection
- Additions need central review

Use a registry

- Independent teams add providers
- Startup explicitly loads extensions
- Duplicate keys have a policy
- Tests isolate the registry

**Registration updates shared state.
Keep it in a deliberate startup step.**

Factory Method

Lets **subclasses choose** which concrete product a creation method returns.

How it works

- The base class uses a creation method to obtain a product.
- Subclasses override that method; products share a common interface.

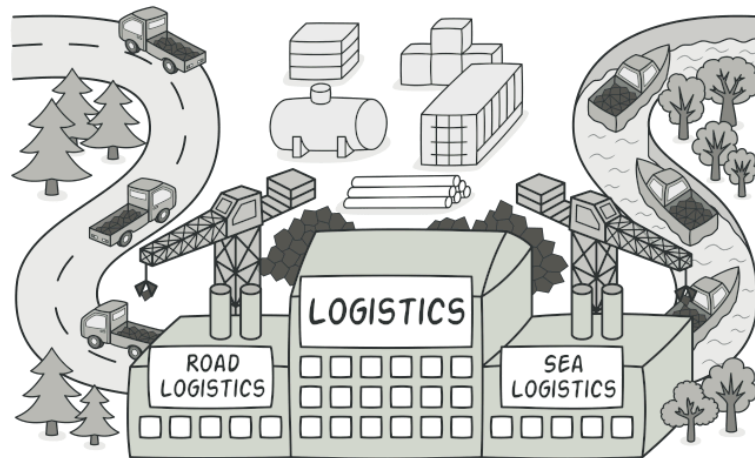


Illustration: [Refactoring.Guru](https://refactoring.guru/) / Dmitry Zhart

Note: A function that selects a class with an if statement is a simple factory; it does not use subclass overrides.

A fresh workspace for each build

```
class BuildWorkflow:
    def run(self, source):
        validate(source)
        with self.make_workspace() as workspace:
            checkout(source, workspace)
            return compile_and_archive(workspace)

    def make_workspace(self):
        raise NotImplementedError

class TemporaryBuild(BuildWorkflow):
    def make_workspace(self):
        return TemporaryDirectory()
```

Before publishing a release, our tool must build an archive. Each build needs a fresh working directory.

- 1 The base workflow fixes the sequence: validate, check out, compile.
- 2 The creation hook runs after validation. Each build gets a fresh workspace.
- 3 TemporaryBuild overrides the hook to choose a temporary directory.
- 4 For TemporaryBuild, with ensures cleanup even if the build fails.

Other builds: BuildVolumeBuild ,
DebugBuild

Abstract Factory

Creates **families of related objects** without exposing their concrete classes to callers.

How it works

- A factory interface defines creation methods for related product types.
- Each concrete factory supplies the products for one compatible family.

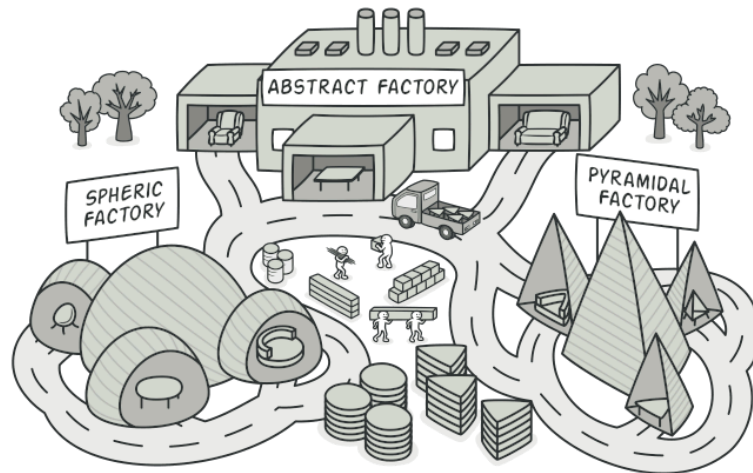
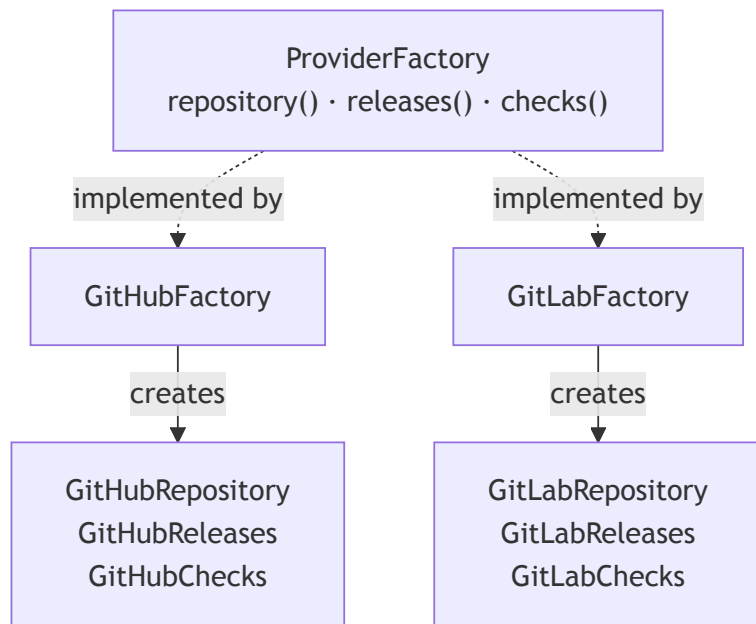


Illustration: [Refactoring.Guru](https://refactoring.guru/) / Dmitry Zhart

Note: Switching factories changes the product family. Adding a new product type requires changing the factory interface.

Abstract Factory creates compatible products



```
class GitHubFactory:
    def repository(self, slug):
        return GitHubRepository(slug)

    def releases(self, repo):
        return GitHubReleases(repo)

    def checks(self, repo):
        return GitHubChecks(repo)

# Client receives the selected factory.
repo = factory.repository("acme/widget")
releases = factory.releases(repo)
checks = factory.checks(repo)
```

One selected factory keeps the provider family consistent.

Separate callers need products from the same provider

```
repo = factory.repository("acme/widget")
releases = factory.releases(repo)
checks = factory.checks(repo)

publisher = ReleasePublisher(releases) # needs release operations
monitor = ReleaseMonitor(checks)      # needs status checks
```

- The publisher uploads releases. The monitor checks their status independently.
- Each caller receives the capability it needs.
- One selected factory creates both products for the same repository and provider.
- If one service handles both jobs through one interface, a single **Provider** adapter may be enough.

Should I use a factory?

Use one when

- Settings choose which class to create
- A workflow needs a new object at a specific step
- Separate callers need objects that work together

Weigh the cost

- Creation code adds another layer to understand and test
- Factory Method subclasses depend on the base workflow
- Abstract Factory adds methods for each product type

If a direct constructor or supplied object is good enough, start there.

Match the mechanism to the requirement

Requirement	Start here
The caller already knows the dependency	Supply an object
Configuration chooses one implementation	Factory function
Independent extensions add implementations	Registry of factories
A workflow needs a fresh object at a particular step	Supplied factory or Factory Method
Callers need separate, compatible products	Abstract Factory

Choose the simplest option that meets the requirement.

Builder, lifetime, and injection

Construction choices also define validation, sharing, and cleanup.

Builder

Separates **step-by-step construction** from the object being built.

How it works

- The builder holds work in progress as callers request construction steps.
- After those steps, the caller obtains the finished product from the builder.

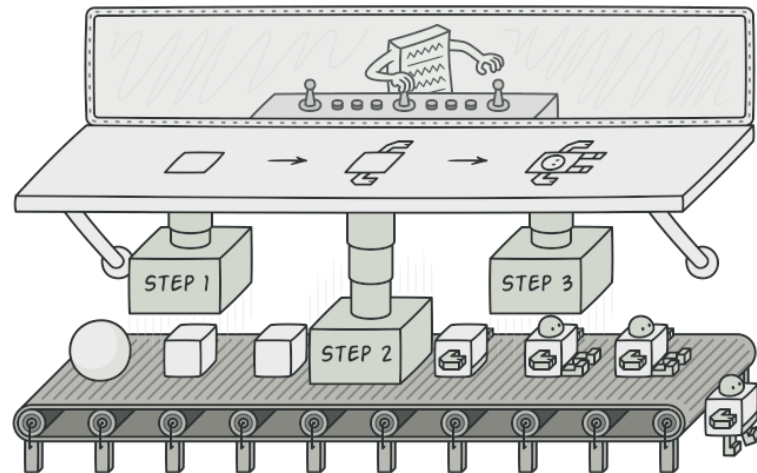


Illustration: [Refactoring.Guru](https://refactoring.guru/) / Dmitry Zhart

Different builders can use the same construction steps to produce different representations.

What does publishing need?

```
from dataclasses import dataclass

@dataclass(frozen=True)
class ReleaseSpec:
    tag: str
    notes: str = ""
    draft: bool = False
    files: tuple[str, ...] = ()
    prerelease: bool = False
```

Repository settings

Start with configured values.

Explicit CLI options

Override selected fields.

Asset paths

Collect files as they are found.

Inputs arrive in stages. One specification holds the result.

Prefer immutable values by default

```
spec = ReleaseSpec(  
    tag="2.1.0",  
    files=("dist/app.zip",),  
)  
  
spec.tag = "2.2.0"  
# FrozenInstanceError  
  
spec.files.append("debug.log")  
# AttributeError
```

Stable during publishing

Create, upload, and complete use the same values.

Safe to share

Callers cannot change each other's inputs.

Collect with a mutable builder. Share an immutable result.

Collect inputs as they become available

```
from dataclasses import dataclass, field

@dataclass
class ReleaseSpecBuilder:
    tag: str = ""
    notes: str = ""
    draft: bool = False
    prerelease: bool = False
    assets: list[str] = field(
        default_factory=list,
    )
```

```
builder = ReleaseSpecBuilder()
builder.from_config(repo_config)
builder.override_with(cli_args)
builder.add_assets(["dist/app.zip"])
spec = builder.build()
```

Only build() validates and returns the completed specification.

Each step updates the same builder

```
# Methods of ReleaseSpecBuilder:
def from_config(self, config):
    for name, value in config.items():
        setattr(self, name, value)
    return self

def override_with(self, explicit_args):
    for name, value in explicit_args.items():
        setattr(self, name, value)
    return self

def add_assets(self, paths):
    self.assets.extend(paths)
    return self
```

setattr(self, name, value)

Set the named field; omitted fields keep their values.

return self

Return this builder so the next call can be chained.

Predict the built value

```
repo_config = {"draft": True, "tag": "2.1.0"}  
cli_args = {"draft": False} # explicit override  
spec = (  
    ReleaseSpecBuilder()  
    .from_config(repo_config)  
    .override_with(cli_args)  
    .add_assets(["dist/app.zip"])  
    .build()  
)
```

What are `spec.draft`, `spec.tag`, **and** `spec.files` ? What if the CLI omits `draft` entirely?

Validate before returning the final value

```
# Final method of ReleaseSpecBuilder:
def build(self) -> ReleaseSpec:
    if not self.tag:
        raise ValueError("a release tag is required")
    if "-rc" in self.tag and not self.prerelease:
        raise ValueError("release candidates must be prereleases")
    return ReleaseSpec(
        tag=self.tag, notes=self.notes, draft=self.draft,
        prerelease=self.prerelease,
        files=tuple(self.assets),
    )
```

1 Reject a missing tag or a release candidate without prerelease=True.

2 Copy the asset list into a tuple so later builder edits cannot change it.

Pass the built specification to the service

```
class ReleaseService:
    ...
    def publish(self, spec: ReleaseSpec) -> None:
        self._provider.create_release(spec)
        for path in spec.files:
            self._provider.upload(path)
        self._provider.mark_complete(spec.tag)
```

- `publish(spec)` replaces `publish(tag, files)` in the remaining examples.
 - we can now accept extra arguments in the future without rewriting call sites
- The provider's `create_release` method now accepts the specification too.

Sometimes a constructor is enough

```
spec = ReleaseSpec(  
    tag=tag,  
    notes=notes,  
    draft=True,  
)
```

All values are available in one place.

Tip: Use named parameters

They explain each value. The constructor can also validate inputs and return an immutable object.

Use a Builder when collecting inputs in stages needs a separate object.

Prototype

Creates new objects by **copying existing ones** through a common interface.

How it works

- The caller asks an existing object for a copy.
- The copy starts with the original's state and can then be modified.

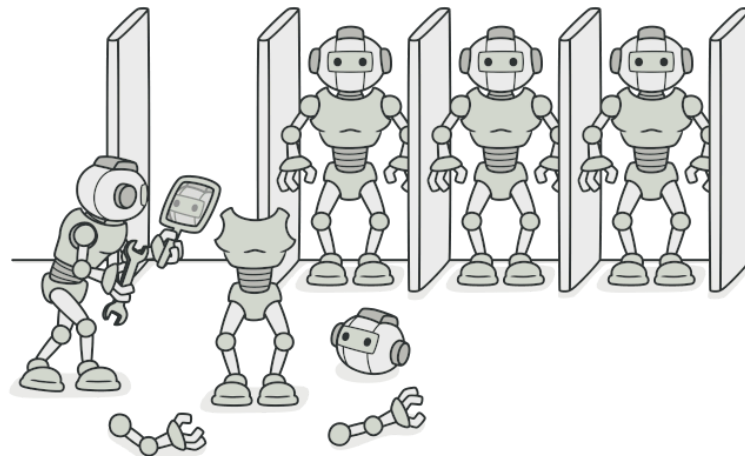


Illustration: [Refactoring.Guru](https://refactoring.guru/) / Dmitry Zhart

Note: Copying must define which nested objects are shared and which are copied. A shallow copy still shares nested objects.

Prototype: copy a configured value

```
from dataclasses import replace

stable_preset = ReleaseSpec(
    tag="2.1.0",
    files=("dist/app.zip",),
)
candidate = replace(
    stable_preset,
    tag="2.1.0-rc1",
    prerelease=True,
)
```

Reuse the preset's fields and change only the release tag and prerelease flag.

A new specification

candidate is a different object.
stable_preset still has tag 2.1.0 and prerelease=False.

The same files tuple

Both specifications refer to the same tuple. Neither can change its contents.

Watch for mutable fields

If files were a list, editing that shared list would affect both objects. This is a shallow copy.

Singleton



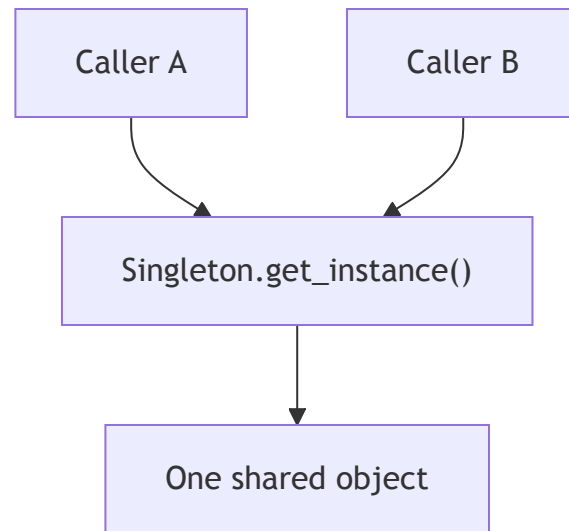
One instance

The class controls construction.



Global access

Every caller retrieves the same object.



One instance per process

```
# clients.py – initialized on first import in each process  
github = GitHubClient(os.environ["GITHUB_TOKEN"])
```



8 WORKER PROCESSES

8 clients

Each process initializes its own object.



2 CUSTOMERS · 1 PROCESS

Same credential

Both use the shared client's token.



2 TESTS · 1 PROCESS

Same state

Changes can carry into the next test.

One request changes another's credential

```
# Both requests received the same mutable client.  
client_a = shared_client  
client_b = shared_client  
  
client_a.token = "customer-A-token" # A starts  
client_b.token = "customer-B-token" # B runs before A publishes  
  
# A resumes, but the shared token now belongs to B.  
client_a.create_release(spec_a)      # uses customer B's token
```

Passing the same client to both requests shares its token too.

Separate creation, lifetime, and access



CREATION

How many?

Create one HTTP client to reuse connections.



LIFETIME

Owned by whom?

An application owns it and closes it at shutdown.



ACCESS

How do callers get it?

Pass the client to each consumer explicitly.

Reusing one object does not require global access.

Sharing and global lookup have different costs

Sharing one instance



One setup

Reuse an expensive client or connection pool.



Coordinate ownership

Choose who can change or close the client.

Looking it up globally



Convenient access

Consumers retrieve the object themselves.



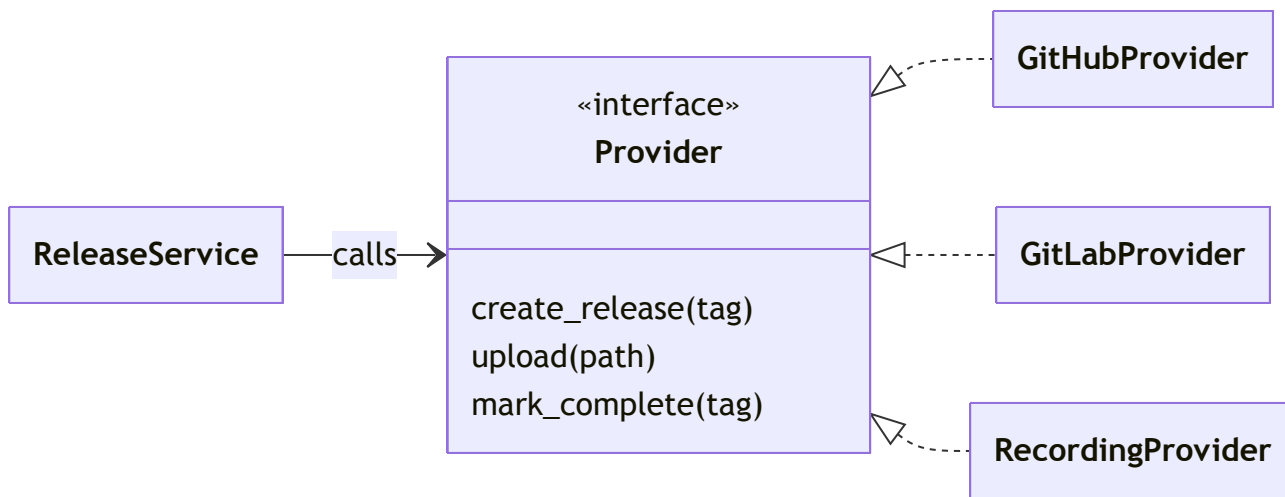
Hidden dependency

Tests need a way to replace that lookup.

Keep reuse. Give the object an owner and pass it explicitly.

Managing Dependencies

Injection makes the provider replaceable



Startup creates and connects the objects

```
def main(settings, spec):  
    load_provider_plugins()  
    with HttpClient() as http:  
        provider = provider_for(  
            settings.provider, http,  
        )  
        service = ReleaseService(provider)  
        service.publish(spec)
```

Composition root

The startup code that creates and connects the application's objects.

- 1 Load provider types and select one.
- 2 Pass dependencies to the provider and service.
- 3 Own the HTTP client's lifetime.

Close the shared client even when publishing fails

```
with HttpClient() as http:
    provider = provider_for(
        settings.provider, http,
    )
    ReleaseService(provider).publish(spec)

# HTTP client is closed.
```

Owner: main()

Creates the client and closes it when the command leaves the `with` block.

Borrower: provider

May use the client. Must not close it.

Success or exception, the HTTP client is always closed.

Test behavior without a network

Fake: fail during upload

```
class FailingProvider:
    def __init__(self):
        self.completed = []

    def create_release(self, spec):
        pass

    def upload(self, path):
        raise RuntimeError("upload failed")

    def mark_complete(self, tag):
        self.completed.append(tag)
```

Test: completion never runs

```
def test_upload_failure():
    provider = FailingProvider()
    spec = ReleaseSpec(
        tag="2.1.0", files=("app.zip",),
    )
    service = ReleaseService(provider)
    with pytest.raises(
        RuntimeError,
        match="upload failed",
    ):
        service.publish(spec)
    assert provider.completed == []
```

Pass the provider directly or look it up?

Receive the provider

```
class ReleaseService:
    def __init__(
        self, provider: Provider,
    ):
        self._provider = provider
```

Visible dependency

The constructor tells callers exactly what to supply.

Look up the provider

```
class ReleaseService:
    def __init__(self, services):
        self._provider = services.get(
            "release_provider"
        )
```

Hidden dependency

The constructor asks for `services`; callers must read its body to learn that it needs `"release_provider"`.

A **service locator** is an object that other objects query for dependencies. Here, `services` is the locator.

Write down ownership and lifetime

Object	Created by	Scope / lifetime	Cleanup
HTTP client	App root	Command / process / tenant	App root closes it
Provider adapter	App root	One release	None here
Release service	Root / request handler	One release here	None here
Release spec	Command handler	One operation	None

Mini Studio

What distinguishes each pattern?

Pattern	Creation mechanism
Factory Function	Function chooses and creates an implementation
Factory Method	Subclass overrides a creation method
Abstract Factory	Factory supplies a family of products
Builder	Builds a product in steps
Prototype	Copies an existing object
Singleton	Controls one globally accessible instance

“Preview complete.” The files were uploaded.

--preview skipped going public, but still uploaded the confidential build.

In groups of 2 to 3, work through the Release Preview handout.

Trace the code, compare the proposals, and recommend a design.



Before you leave: Show your packet to a TA

