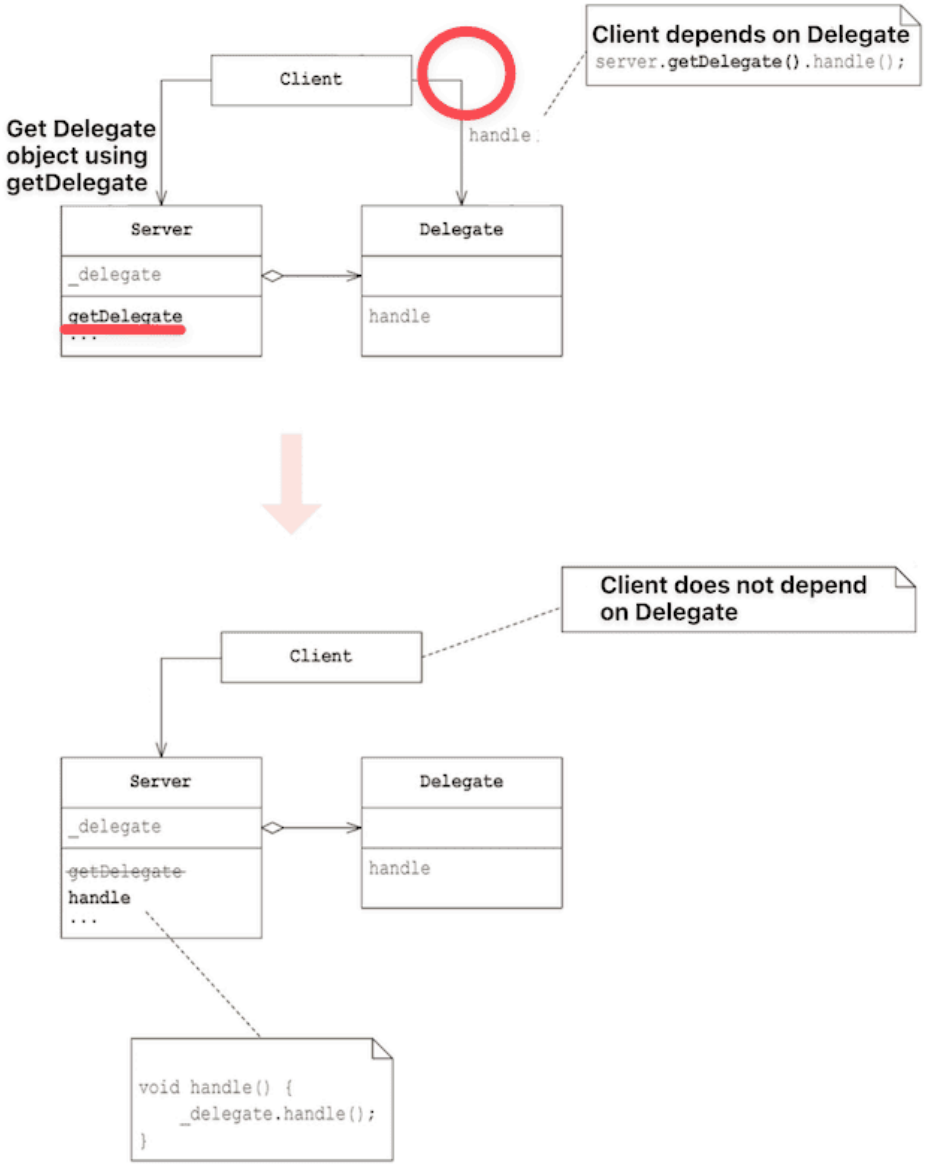
Hide Delegation

Reduce coupling by **adding wrapper methods** so clients don’t access delegate objects directly.

- When modules are interacting with too many other modules, it is hard to manage them.
- In this case, we use the Hide Delegation refactoring to remove dependencies.

- This is also called the `law of demeter (LoD).`
- This design rules shows the importance of removing direct coupling as in this example.


Client depends on Delegate
server.getDelegate().handle();
Client
Get Delegate object using getDelegate
handle
Server
_delegate
getDelegate
...
Delegate
handle
Client does not depend on Delegate
Client
Server
_delegate
getDelegate
handle
...
Delegate
handle
void handle() {
_delegate.handle();
}

Example: Database

- We have an address management system (AddressFile) that uses a database (Database).
- For database, we use a Python dictionary.

Database

- The Python dictionary contains properties.
- The load method reads the database content from a file.

```
class Database:
    def __init__(self, filename: str):
        self.filename = filename
        self.properties: Dict[str, str] = {}
        self.load()
```

```
def load(self):
    if os.path.exists(self.filename):
        try: ...
        except IOError:
            pass
def set(self, key: str, value: str):
    self.properties[key] = value

def get(self, key: str) -> Optional[str]:
    return self.properties.get(key)

def update(self):
    with open(self.filename, 'w', encoding='utf-8') as f:
        f.write("#\n")
        for key, value in self.properties.items():
            f.write(f"{escaped_key}={value}\n")
def get_properties(self) -> Dict[str, str]:
    return self.properties
```

AddressFile

- The names method returns an iterator to access database keys.
- This is a code smell that introduces coupling.

```
class AddressFile:
    def __init__(self, filename: str):
        self.database = Database(filename)
    def get_database(self) -> Database:
        return self.database
    def names(self) -> Iterator[str]:
        return iter(self.database.get_properties().keys())
```

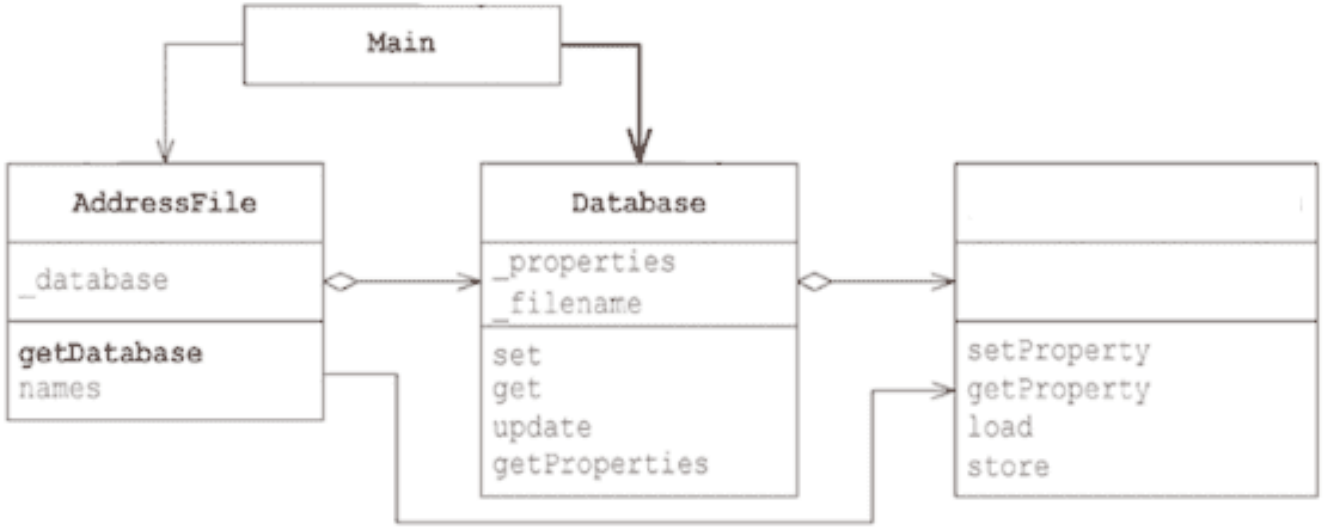
Main

- The main method reads the file and updates the database.

```
def main():
    try:
        file = AddressFile("address.txt")
        # Using exposed database - this shows the problem
        # ????
        file.get_database().set("John Deacon", "deacy@example.com")
        # ???
        file.get_database().update()
        for name in file.names():
            mail = file.get_database().get(name)
            print(f"name={name}, mail={mail}")
    except Exception as e:
        print(f"Error: {e}")
```

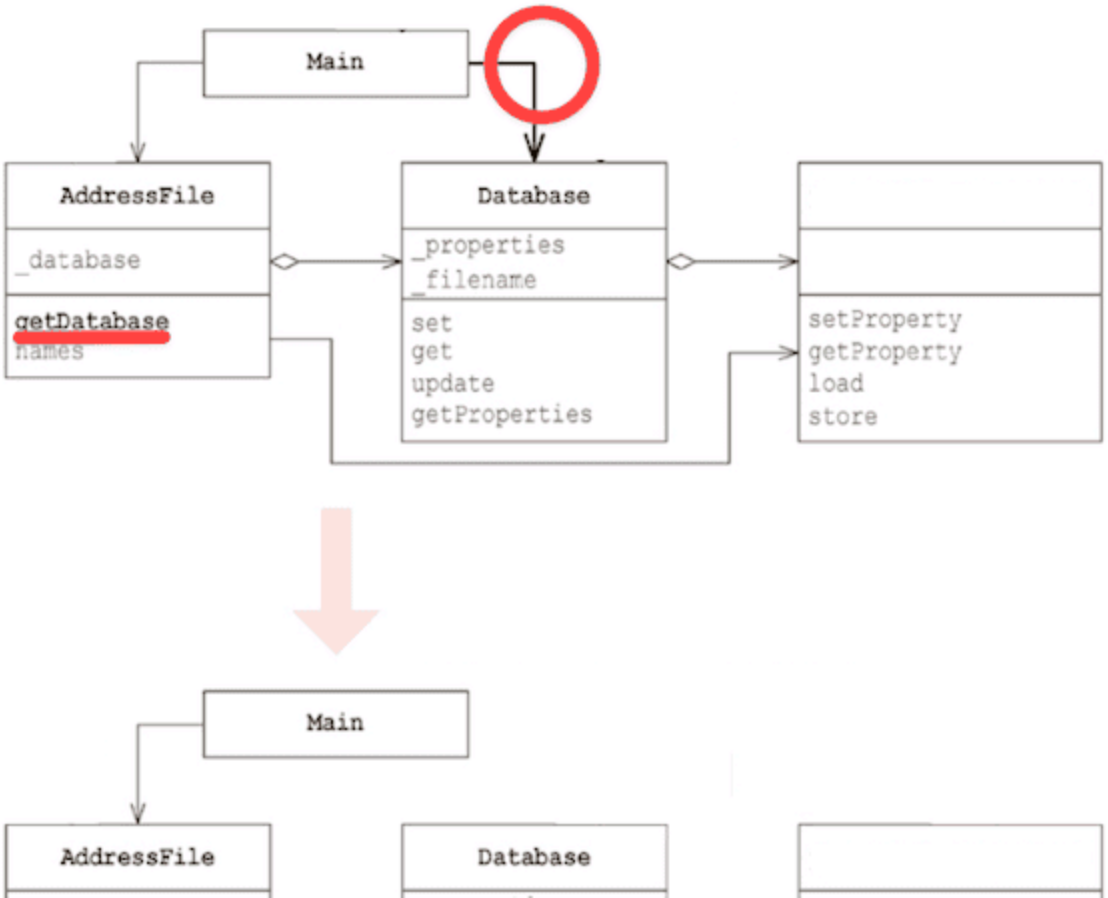
UML

- The main method knows both the AddressFile and Database.



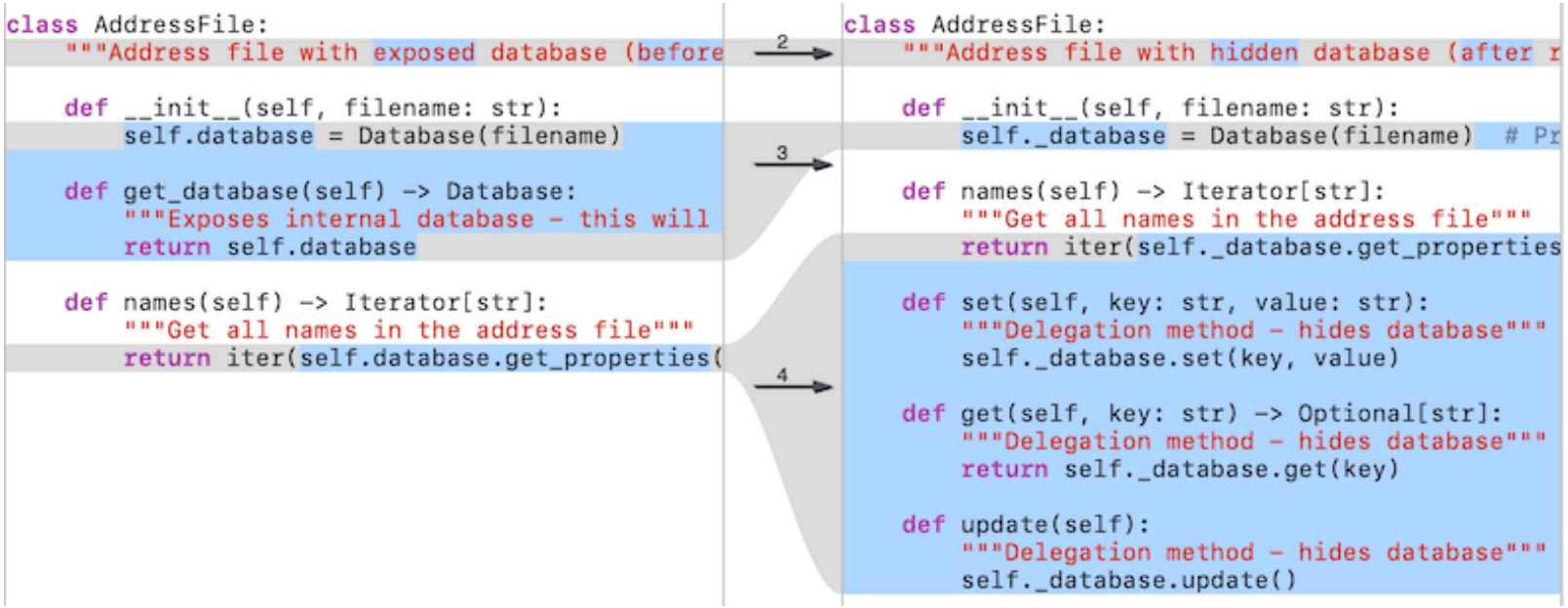
Refactoring: Hide Database

- The first step is to remove the coupling between Main and Database.



- We need to prevent the direct access of the Database from the main.
- Instead, we provide public APIs to access database.

- Main can use set/get method to access database indirectly.



```
def main():
    try:
        file = AddressFile("address.txt")
        # Using delegation methods - database is hidden
        file.set("John Deacon", "jd@example.com")
        file.set("Brian May", "may@example.com")
        file.set("Roger Taylor", "rtaylor@example.com")
        file.update()
```

Refactoring Again: Remove Coupling

The next step is to remove the coupling between AddressFile and Dict.

Database

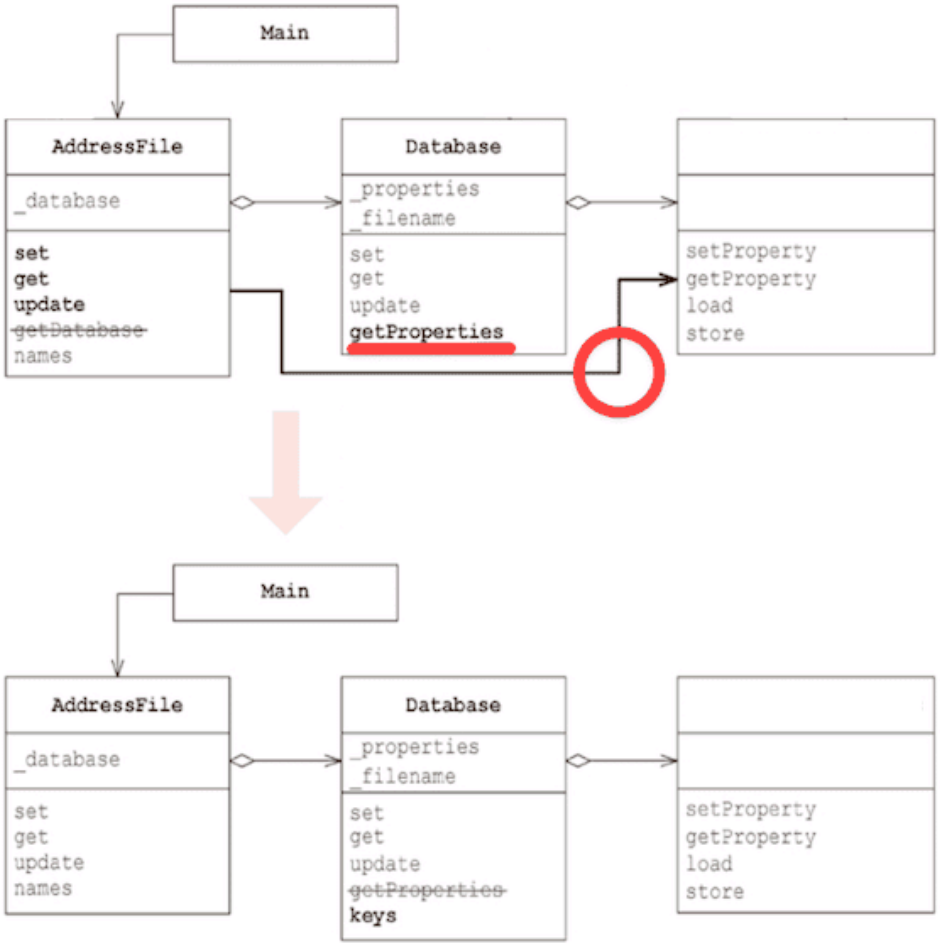
- We can get iterator over keys.
- It is an alternative to the getProperties().

```
def keys(self):
    return iter(self.properties.keys())
```

AddressFile

- We get all names using Database.keys(), not direct access of Database.properties.

```
def names(self):
    return self._database.keys()
```


Main
AddressFile
_database
set
get
update
getDatabase
names
Database
_properties
_filename
set
get
update
getProperties
setProperty
getProperty
load
store
Main
AddressFile
_database
set
get
update
names
Database
_properties
_filename
set
get
update
getProperties
keys
setProperty
getProperty
load
store

Information Hiding

- We need to hide information as much as possible to avoid coupling.
- We provide public APIs to hide provide an access to hidden methods or fields.

Discussion

Hide Delegate reduces coupling, but if overused, it creates a "Middle Man" class full of wrapper methods.

1. When you have **too many wrapper methods** — server class becomes a "middle man"
2. When the server class becomes **bloated with middle man logic** to hide delegation

Benefits of Hide Delegate

1. **Reduced coupling** - clients don't depend on delegate structure
2. **Encapsulation** - internal relationships are hidden
3. **Easier changes** - can modify delegate structure without affecting clients
4. **Simpler client code** - clients use cleaner, more focused interface
5. **Better abstraction** - server class provides appropriate level of abstraction