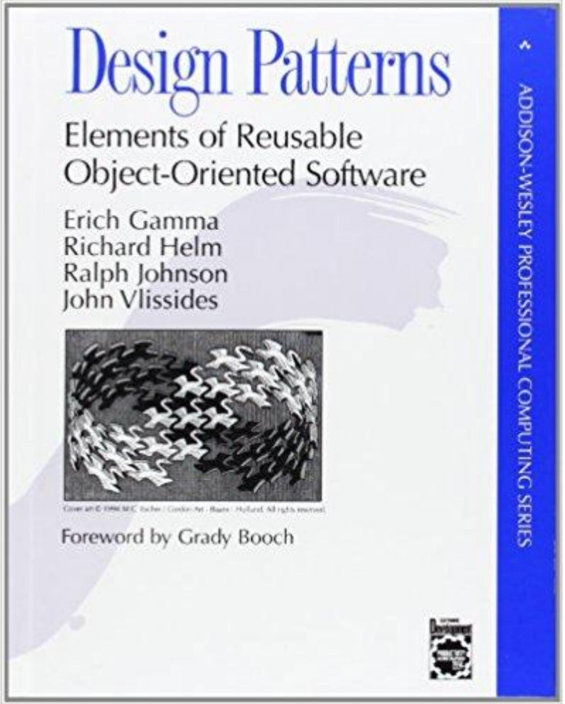
Design Patterns

Don’t Reinvent the Wheel!



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

One-slide summary: Why do we need design patterns?

- **Design patterns** separate the **structure** of a system from **implementation details.**
- Design is hard!
- Every design has tradeoffs
  - Software design patterns often trade more verbosity or less efficiency for easier extensibility.
- We'll look at examples of **creational, behavioral, and structural** software design patterns.

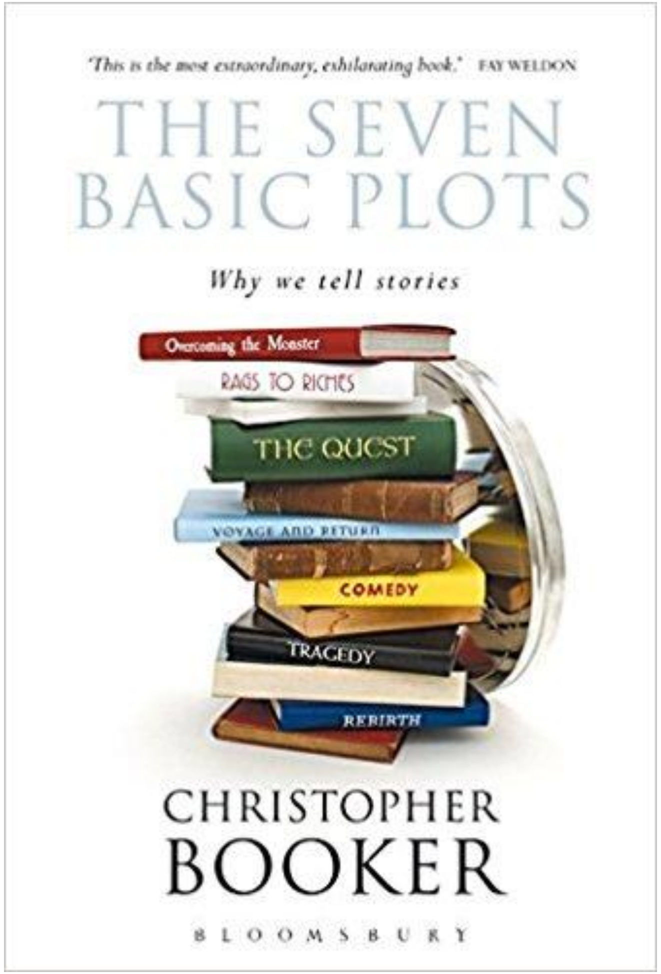
Boring Technical Definition

“A general, reusable solution to a commonly occurring problem within a given context.”

- Wikipedia

Design patterns are everywhere!

- Musical forms and structures
- Exam questions (sorry)
- Road systems
- Storylines



Design patterns vs. idioms?

- What’s the difference?
  - Depends on who you ask.
- Two points on the same spectrum.
- This lecture will include some “proper” design patterns and some idioms.

Creational Patterns

- When is a constructor not good enough?
  - Control how an object is created.
  - Language limitations, i.e. lack of keyword and default arguments.
  - Need to hide polymorphic types from user.

Named Constructor (Idiom)

- Foundation for other creational patterns.
- Make constructor private, expose static “create” method.

```
class Llama {
public:
    static Llama* create_llama() { ... }

private:
    Llama() {}
};
```

Factory Pattern

- **Problem**: We want to create polymorphic objects without exposing their types to the client.
- **Option 1**: A function that takes in a string indicating which type to create.

```
Llama* llama_factory(string name, string type) {
    if (type == “ninja_llama”) {
        return new NinjaLlama(name);
    }
    else if (type == “whooping_llama”) {
        return new WhoopingLlama(name);
    }
}
```

Factory Pattern

- **Option 2**: Create a class with one static method per object type.

```
class LlamaFactory {
public:
    static Llama* make_ninja_llama(string name) {
        return new NinjaLlama(name);
    }
    static Llama* make_whooping_llama(string name) {
        return new WhoopingLlama(name);
    }
};
```

Factory Pattern: Let’s make it fancier

- **Scenario:**
  - You’re implementing a computer game with some Enemy class hierarchy.
  - Want to change enemy strategies based on difficulty setting.
- **Bad solution:**

```
class Enemy {
    void attack() {
        if (difficulty == “easy”) { … }
        else if (difficulty == “hard”) { ... }
        ...
    }
};
```

Solution: Abstract Factory Pattern

- Make the factory class polymorphic.

Solution: Abstract Factory Pattern

- Make the factory class polymorphic.

```
class AbstractEnemyFactory {
public:
    virtual Enemy* create_goomba() = 0;
};
```

```
class Enemy {...};
class Goomba: public Enemy {...};
class SuperGoomba: public Goomba {...};
```

Solution: Abstract Factory Pattern

- Make the factory class polymorphic.

```
class AbstractEnemyFactory {
public:
    virtual Enemy* create_goomba() = 0;
};
class EasyEnemyFactory {
public:
    Enemy* create_goomba() override {
        return new Goomba;
    }
};
```

```
class Enemy {...};
class Goomba: public Enemy {...};
class SuperGoomba: public Goomba {...};
```

Solution: Abstract Factory Pattern

- Make the factory class polymorphic.

```
class AbstractEnemyFactory {
public:
    virtual Enemy* create_goomba() = 0;
};
class EasyEnemyFactory {
public:
    Enemy* create_goomba() override {
        return new Goomba;
    }
};
class HardEnemyFactory {
public:
    Enemy* create_goomba() override {
        return new SuperGoomba;
    }
};
```

```
class Enemy {...};
class Goomba: public Enemy {...};
class SuperGoomba: public Goomba {...};
```

Solution: Abstract Factory Pattern

- Make the factory class polymorphic.

```
class AbstractEnemyFactory {
public:
    virtual Enemy* create_goomba() = 0;
};
class EasyEnemyFactory {
public:
    Enemy* create_goomba() override {
        return new Goomba;
    }
};
class HardEnemyFactory {
public:
    Enemy* create_goomba() override {
        return new SuperGoomba;
    }
};
```

```
class Enemy {...};
class Goomba: public Enemy {...};
class SuperGoomba: public Goomba {...};


int main() {
    AbstractEnemyFactory* factory = nullptr;
    if (difficulty == “easy”) {
        factory = new EasyEnemyfactory;
    }
    else if (difficulty == “hard”) {
        factory = new HardEnemyFactory;
    }
    ...
    factory->create_goomba();
}
```

Managing global state

- **Scenario**:
  - Have application state that needs to be globally accessible.
  - Need controlled read/write access for that data.
- **Bad solution**: Naked global variables
  - No. Just no.
- **Less bad solution**: Write a class, have one global instance of it.
  - Less global namespace clutter.
  - But that one global instance is still naked.

Solution: Singleton Pattern

- Ensure that exactly one instance of a class exists.
  - That instance should be available when requested.
- Control access to that instance.
- Implementation:
  - Make constructor private (like Named Constructor!)
    - Note: We’ll be using Python from here on, so we’ll just pretend…
    - Singleton pattern in C++ is complicated by lack of garbage collection.
      - Further reading: http://www.umich.edu/~eecs381/lecture/IdiomsDesPattsCreational.pdf
  - Expose static “get_instance” or “get” method (these slides call it “get”).
  - “get” method creates the instance if it doesn’t exist yet.

Singleton Pattern

```
class Singleton:
    @staticmethod
    def get():
        ...

    _instance = None
```

Singleton Pattern

```
class Singleton:
    @staticmethod
    def get():
        if Singleton._instance is None:
            Singleton._instance = Singleton()
        return Singleton._instance

    _instance = None
```

Singleton Pattern

```python
class Singleton:
    @staticmethod
    def get():
        if Singleton._instance is None:
            Singleton._instance = Singleton()
        return Singleton._instance

    _instance = None

    def __init__(self):
        self._spams = 42
        print("Singleton created")

    def num_spams(self):
        return self._spams
```

Singleton Pattern

```python
class Singleton:
    @staticmethod
    def get():
        if Singleton._instance is None:
            Singleton._instance = Singleton()
        return Singleton._instance

    _instance = None

    def __init__(self):
        self._spams = 42
        print("Singleton created")

    def num_spams(self):
        return self._spams
```

```python
def main():
    # The "created" message should
    # only print once
    print(Singleton.get().num_spams())
    print(Singleton.get().num_spams())
```

Exercise: LlamaFarm

- Write a singleton class, **`LlamaFarm`** that:
  - Stores a list of `Llamas`.
  - Has a method **`add_llama`** that adds a `Llama` object to the list.
- Modify the **`Llama`** class below so that:
  - Whenever a `Llama` is created, it automatically is added to the `LLamaFarm`.

Starter code is available at: goo.gl/sq8atz

Exercise: LlamaFarm (Discussion)

- I used this technique for my EECS 494 (game design) project, “Ninja Brian and Steve the Llama”.
- Many enemies hand-placed in the scene.
- Needed them automatically added to a list for lock-on system.
- Is this something you should do all the time? NO!!
  - In later examples in this lecture, we’ll take a different approach.
  - Always consider the tradeoffs within your application’s constraints.

Singleton: Tradeoffs and Pitfalls

- Is it OK to call **get()** once and store the reference in a variable?
    - Nope! Singleton only guarantees one instance at a time.
    - That instance could be replaced, as long as there’s only one at a time.
    - Access to the instance is only guaranteed by calling **get()**.
- How could the Singleton pattern be abused?
    - “Let’s put ALL our data there!”

Behavioral Patterns

“Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.”

- “Gang of Four” Design Patterns book

- One such pattern you’ve seen and implemented: **Iterator pattern**
    - Provides a uniform method for traversing containers without knowing how they’re implemented

Observer Pattern

- **General idea:**
  - We have classes of objects that need to stay up to date with certain changes in the program.
  - Need some way of tracking which objects need to be updated when a change happens.
  - The object that changed shouldn’t need to know details about the objects that need to be updated.

Observer Pattern

- **Scenario 1**:
  - Autograder website
  - We need to display a “loading” overlay while certain actions are pending.
- **Scenario 2**:
  - Lock-on targeting system in “Ninja Brian and Steve the Llama”
  - When a locked-onto enemy is defeated, player should automatically target another one

Exercise: Observant Llamas

- Update `Llama` and `LlamaFarm` so that:
    - `Llamas` keep track of a list of their friends.
    - When a `Llama` is added to the `LlamaFarm`, the other `Llamas` add it as a friend.

Starter code is available at: goo.gl/sq8atz

Model-View-Controller

- Or if you prefer: Singleton + Observer + A cool new hat.
- **Goal:** Separate the “business logic” of an application from it’s user interface.
- Ubiquitous in GUI and web frameworks.

Model

- Manages the “business logic” of the application.
- Store data, expose methods for accessing and modifying it.
- In our examples, the Model will be a Singleton class.
- In other applications, could be a database or web server.
- The “subject” in the Observer pattern.

View

- Displays information to the human user.
- The Model should NOT know about the concrete View classes.
- We should be able to add and remove Views from our hierarchy without changing the Model.
- Needs to receive data from the Model. Two options:
  - **Push:** `update()` functions pass in the updated data as parameters.
  - **Pull:** `update()` functions don't pass parameters, View must call some Model method to get the new data after receiving the update notification.

View

- What are the tradeoffs of push vs pull?
  - Generally, push means simpler Model data accessors, but more fine-grained `notify()` and `update()` functions needed.
  - Similarly, pull means simpler `notify()` and `update()` functions, but more complicated Model data accessors.
- Weigh your application’s constraints!

Controller

- Processes commands from the human user, i.e.:
  - Perform some action on the Model.
  - Change the current view.
- In GUI applications, usually tightly coupled with Views
  - User clicks a button on the page, which changes the Model’s data.
  - Model pushes the changed data to the View, which updates what the user sees.

Live Coding: LlamaFarm MVC

Template Method Pattern

- Define the structure of an algorithm in a base class.
- Derived classes can override specific steps of the algorithm.
- *Very* useful in libraries and frameworks.

Template Method Pattern: Key Detail

- The algorithm defined by the base class is a **non-virtual function!**
- The steps of the algorithm that can be overridden are virtual!
- The “Hollywood Principle”
    - Base class method calls derived class methods.
    - “Backwards” from derived class methods calling base class methods.

Template Method Pattern: Example

```
// C++-ish pseudocode
class HTTPRequestHandler {
public:
  handle_request(request) {
  }
}
```

Template Method Pattern: Example

```
// C++-ish pseudocode
class HTTPRequestHandler {
public:
  handle_request(request) {
  }
protected:
  virtual User authenticate(request) {...}
  virtual Data load_data(request) {...}
  virtual check_permissions(data, user) {...}
}
```

Template Method Pattern: Example

```
// C++-ish pseudocode
class HTTPRequestHandler {
public:
  handle_request(request) {
    try {
    }
    catch(AuthError&) {...}
    catch(PermissionError&) {...}
  }
protected:
  virtual User authenticate(request) {...}
  virtual Data load_data(request) {...}
  virtual check_permissions(data, user) {...}
}
```

Template Method Pattern: Example

```
// C++-ish pseudocode
class HTTPRequestHandler {
public:
  handle_request(request) {
    try {
      user = authenticate(request)
      data = load_data(request)
      check_permissions(data, user)
      return Response(data)
    }
    catch(AuthError&) {...}
    catch(PermissionError&) {...}
  }
protected:
  virtual User authenticate(request) {...}
  virtual Data load_data(request) {...}
  virtual check_permissions(data, user) {...}
}
```

Template Method Pattern: Example

```
// C++-ish pseudocode
class HTTPRequestHandler {
public:
  handle_request(request) {
    try {
      user = authenticate(request)
      data = load_data(request)
      check_permissions(data, user)
      return Response(data)
    }
    catch(AuthError&) {...}
    catch(PermissionError&) {...}
  }
protected:
  virtual User authenticate(request) {...}
  virtual Data load_data(request) {...}
  virtual check_permissions(data, user) {...}
}
```

```
class SanicRequestHandler:
    public  HTTPRequesthandler {
protected:
  Data load_data(request) {
    // Load the data FAST!
  }
}
```

Structural Patterns

- Build up new classes and interfaces from existing ones.
- Hide implementation details from a user.
- Provide a cleaner or more specialized interface.
- You’ve seen some of these before in........EECS 280!

Adapter/Wrapper Pattern

- Apple’s favorite design pattern apparently…
- In EECS 280, you:
  - Implemented a Stack that was an adapter for a linked list.
    - “Shrunk” or specialized the available interface
  - Implemented a Map that was a wrapper for a binary search tree.
    - Expanded the interface and functionality

Adapter/Wrapper Pattern (More examples)

- Early implementations of fstreams in C++ were just wrappers around the C FILE macro.
- Autograder: I wrote a wrapper/adapter around a containerization library.
  - Irons out not-so-fun-quirks of the library
  - Provides specialized interface
- Many other Structural patterns are some variation of this.

Wrapping Up

- Design is hard! Don’t reinvent the wheel!
- Always consider the tradeoffs of design patterns!
- If your application constraints don’t fit a design pattern, then *don’t use a design pattern!*
- Design patterns, object-oriented programming aren’t always the right approach…
- But when they are, use them properly!