CS 4500
Software Development

Design Patterns

October 11, 2019

Motivation

Many design issues / challenges:

- Have been faced before
- Have been addressed and solved repeatedly
- Some solutions are more successful than others
- Abstraction of solutions to design problems = design pattern

Design Patterns

- Goal: not to solve every problem from first principles

Design Patterns

- Goal: not to solve every problem from first principles
- Simple and elegant solutions to specific (recurring) problems in component design

Design Patterns

- Goal: not to solve every problem from first principles
- Simple and elegant solutions to specific (recurring) problems in component design
- A three-part rule – expresses a relation between a certain context, a problem, and a solution

Design Patterns

- Goal: not to solve every problem from first principles
- Simple and elegant solutions to specific (recurring) problems in component design
- A three-part rule – expresses a relation between a certain context, a problem, and a solution
- Captures design knowledge in a way that allows others to use that knowledge

Design Patterns

- Goal: not to solve every problem from first principles
- Simple and elegant solutions to specific (recurring) problems in component design
- A three-part rule – expresses a relation between a certain context, a problem, and a solution
- Captures design knowledge in a way that allows others to use that knowledge
- Originated in OOD, but applicable to other approaches to modular design

Design Patterns

- Goal: not to solve every problem from first principles
- Simple and elegant solutions to specific (recurring) problems in component design
- A three-part rule – expresses a relation between a certain context, a problem, and a solution
- Captures design knowledge in a way that allows others to use that knowledge
- Originated in OOD, but applicable to other approaches to modular design
- Seen some in CS3500

Effective Design Patterns

Solve a problem Capture solutions, not just abstract principles or strategies.

Effective Design Patterns

Solve a problem Capture solutions, not just abstract principles or strategies.

Proven concept Capture solutions with a track record, not theories or speculation.

Effective Design Patterns

Solve a problem Capture solutions, not just abstract principles or strategies.

Proven concept Capture solutions with a track record, not theories or speculation.

Solution isn't obvious Best patterns generate a solution to a problem indirectly

Effective Design Patterns

Solve a problem: Capture solutions, not just abstract principles or strategies.

Proven concept: Capture solutions with a track record, not theories or speculation.

Solution isn't obvious: Best patterns generate a solution to a problem indirectly

Describes a relationship: Patterns don't just describe modules, describe deeper system structures and mechanisms.

Effective Design Patterns

Solve a problem: Capture solutions, not just abstract principles or strategies.

Proven concept: Capture solutions with a track record, not theories or speculation.

Solution isn't obvious: Best patterns generate a solution to a problem indirectly

Describes a relationship: Patterns don't just describe modules, describe deeper system structures and mechanisms.

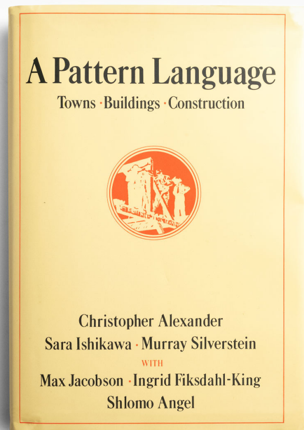
Significant human component: Best patterns explicitly appeal to aesthetics and utility.

A Pattern Language

- DPs originated in architecture
  - but inspired by CS

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

“Each solution is stated in such a way that it gives the essential field of relationships needed to solve the problem, but in a very general and abstract way—so you can solve the problem for yourself, in your own way, by adapting it to your preferences, and the local conditions at the place where you are making it.”



A Pattern Language (1977) – contains 253 patterns of architectural and urban design

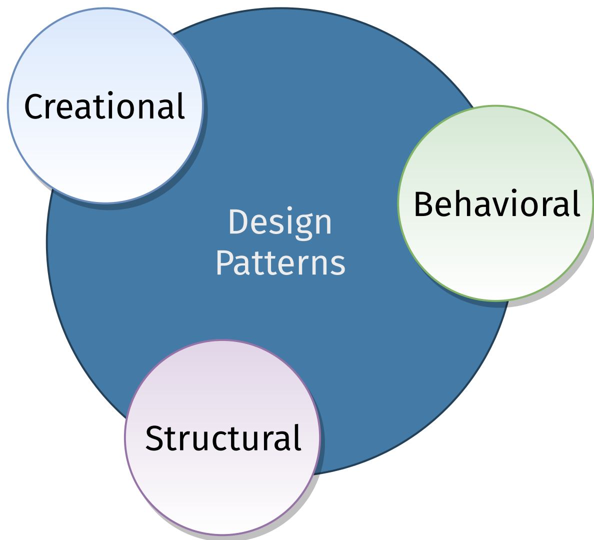
Design Patterns as a Language

- Organization principle for software components
- Guides design of components
- DP catalogs – usually presented using a template:
  - DP name
  - Problem it addresses
  - Motivation – example
  - Applicability
  - Consequences
  - ...

Design Patterns as a Language

- Shared design knowledge
- Recognizable
  - Readability
  - Maintainability
  - Easier communication

Classification



- Also: Concurrent

“Gang of Four” Patterns

Creational

- Abstract Factory
- Builder‡
- Factory Method
- Prototype
- Singleton

Structural

- Adapter‡
- Bridge
- Composite
- Decorator‡
- Facade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- Command‡
- Interpreter
- Mediator
- Memento
- Observer
- State
- Strategy‡
- Template Method
- Visitor

---

‡Possibly seen in CS3500… ?

Creational Patterns

- Abstract the instantiation process of objects
- Make system independent from how its objects are created
- Creation, composition, representation of objects
- Encapsulate all knowledge about which concrete classes the system is using
- Hide how instances of these classes are created and put together

Builder

Example

Constraint: the shape of a Labyrinth is immutable

```
class Labyrinth {
  private Node[] nodes;
  private Edge[] edges;

  public Labyrinth(Collection<Node> nodes, Collection<Edge> edges) { ...
  ...
```

Use:

```
Node nodeA = new Node("A");
Node nodeB = new Node("B");
Node nodeC = new Node("C");

ArrayList<Node> nodes = new ArrayList();
nodes.add(nodeA);
nodes.add(nodeB);
nodes.add(nodeC);
TreeSet<Edge> edges = new TreeSet(
  Arrays.asList(new Edge(nodeA, nodeB), new Edge(nodeB, nodeC)));
Labyrinth lab = new Labyrinth(nodes, edges);
...
```

With a Builder Interface

```
class Labyrinth {
  private Node[] nodes;
  private Edge[] edges;

  private Labyrinth(Node[] nodes, Edge[] edges) {
  public static Builder getBuilder() { return new Builder(); }

  public static class Builder {
    private Edge[] edges;
    private Node[] nodes;

    public Builder();
    public void addNode(String name);
    public void addEdge(String from, String to);
    public Labyrinth build() { ... }
  }

  ...
```

Use:

```
Labyrinth.Builder builder = Labyrinth.getBuilder();
builder.addNode("A");
builder.addNode("B");
builder.addEdge("A", "B");
builder.addNode("C");
builder.addEdge("B", "C");
Labyrinth lab = builder.build();
```

Builder

With Chaining

Alternatively:

```
  public static class Builder {
    ...
    public Builder();
    public Builder addNode(String name);
    public Builder addEdge(String from, String to);
    public Labyrinth build();
  }
  ...
```

Use:

```
Labyrinth lab = Labyrinth.getBuilder()
  .addNode("A")
  .addNode("B")
  .addEdge("A", "B")
  .addNode("C")
  .addEdge("B", "C")
  .build();
```

Builder

- Situation: we have an object that, upon creation, can be set up in complex ways
- Maybe we also have a default configuration, or a few default configurations
- Goal: separate object creation and configuration from its representation

Singleton

```
class Logger {
  public Logger(String filename);
  public void warning(String message);
  public void info(String message);
  ...
}

class Foo {
  public Foo(Logger logger) {
    ... new Bar(logger) ...
  }
```

```
Logger logger = new Logger("log");
Foo foo1 = new Foo(logger);
...
Foo foo2 = new Foo(logger);
```

Singleton

```java
class Logger {
  private Logger instance = null;
  public static Logger getInstance() {
    if (instance == null) {
      this.instance = new Logger("log");
    }
    return this.instance;
  }

  public log(String message) { ... }

  private Logger(String filename) { ... }
}
```

Singleton

```
class Logger {
  private Logger instance = null;
  public static Logger getInstance() {
    if (instance == null) {
      this.instance = new Logger("log");
    }
    return this.instance;
  }

  public log(String message) { ... }

  private Logger(String filename) { ... }
}
```

- Careful about multiple threads

Singleton

- Used to limit to a single instance of a class
- Single global access point to instance
- Unlike `static`, enforces initialization
- E.g.: database connection manager, logging service

Singleton

- Used to limit to a single instance of a class
- Single global access point to instance
- Unlike `static`, enforces initialization
- E.g.: database connection manager, logging service
- Somewhat controversial – global point of entry – might obscure control flow

Structural Patterns

- Focus: how classes and objects are composed to form larger structures
- Help establish relationships between entities within a system
- “Class patterns”: compose interfaces or implementations
- “Object patterns”: compose objects to realize new functionality

Adapter

- Converts between interfaces: Adaptee and Target
- If an object provides a different interface than a client expects
- Two approaches:
    - Object adapter: implements Target's interface by maintaining an instance of Adaptee delegating to it at runtime
    - Class-based: implements Target's interface and inherits from Adaptee at compile-time

Adapter

```
interface Array<T> {
  T get(int index);
}

interface Iterable<T> {
  Iterator<T> iterator();
}

interface Iterator<T> {
  T next();
  T prev();
  T reset();
}

class IterableAsArray<T> implements Array<T>{
  T get(int index) { ... }
}
```

Proxy

- Provide a surrogate or a place holder for another object
- Allows controlling access to the object it represents
- Implements the same interface as the real subject
  - Prevents complicating the client of the real *subject*
  - Clients can't tell if they are using a surrogate
- Scenarios:
  - Delaying the initialization of an expensive resource – on-demand loading – *virtual proxy*
  - Caching for a slow connection
  - Control access based on access rights – *protection proxy*
  - Local representation for a remote object – *remote proxy*
  - Ensuring that a resource is locked before it's accessed – *smart proxy*

```java
public interface Image {
    void display();
}

public class RealImage implements Image {
    public RealImage(String fileName) { loadFromDisk(fileName); }
    public void display() { ... }
    private void loadFromDisk(String fileName) { ... }
}

public class ProxyImage implements Image{
    private RealImage realImage = null;
    public ProxyImage(String fileName) { ... }
    public void display() {
        if (realImage == null)
            realImage = new RealImage(fileName);
        realImage.display();
    }
}
```

Remote Proxy

- Represents a remote resource locally
- Useful, e.g., when switching between a local to a remote implementation
- Hiding that the system is distributed
- Example: local file access → WebDAV
- Example: using a remote service to render an image

Remote Proxy

```
interface Labyrinth {
  public void create(Set<Node> nodes, Set<Edge> edges);
  public void addToken(Node node, Color token);
  public bool reachable(Color token, Node node);
}

class LocalLabyrinth implements Labyrinth { ... }

class TCPLabyrinth implements Labyrinth {
  public TCPLabyrinth(String host, int port) { ... }
  ...
  public void addToken(Node node, Color token) { ... sendRequest; ... }
}
```

Behavioral

- Algorithms
- Assignment of responsibilities between objects
- Patterns of communication between objects
- Characterize complex control flow
- Shift focus: from flow of control toward the way objects are interconnected

Observer

- An object (subject) maintains a list of observers
- Observers get notified of state changes
- Scenarios:
  - Update multiple views of the same data when the data changes
  - Clients perform actions in lock-step and need to be notified of the tick of a common clock
- Careful: avoid a chain of observers

```java
interface Observer<T> {
  void update(T data);
}

interface Subject<T> {
  public void attach(Observer<T> observer);
  public void detach(Observer<T> observer);
  public void notify();
}
```

Applicability / Discussion

- Are common design patterns applicable in your language?
- Are they needed?
- How about different paradigms?

Resources

- Gamma, Helm, Johnson & Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995 – aka the Gang of Four
- Freeman, Robson, Bates, Sierra. *Head First Design Patterns*. 2004
- `https://en.wikipedia.org/wiki/Software_design_pattern`
- `https://refactoring.guru/design-patterns/catalog`

Summary

Design Patterns

- Successful solutions to software design problems – abstracted
- A language for building software components
- Available as repositories of patterns