Introduction to Design Patterns

- What is a software design pattern?
- Examples of software design patterns
- What does a software design pattern look like?
- How is a software design pattern used?

What is a software design pattern?

- A design pattern
  - Is a description of a problem which occurs frequently in various contexts
  - Describes the core of a solution that can be implemented "a million times over, without ever doing it the same way twice"
  - Should:
    - Describe a design abstraction that makes your design more flexible
    - Lead to an overall software design that is easier to change as requirements evolve over time

Software design patterns: Brief History

- Software design patterns became widely discussed in 1995
  - Book: *Design Patterns, Elements of Reusable Object-Oriented Software*
  - By *Gang of Four* (GoF)
    - Inspired by importance of patterns in other disciplines
    - This is particularly true of Christopher Alexander, who documented many patterns for building architecture discipline

Examples of software design patterns

- Brief descriptions of some design patterns from
  - Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.
  - Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Second Edition. Prentice Hall.
  - Fernandez, E.B. (2013). *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley.

GoF Sample Design Patterns

- Organized their patterns by purpose
  - Creational
    - Purpose: object creation
    - Five creational patterns
  - Structural
    - Purpose: deal with composition of classes or objects
    - Seven structural patterns
  - Behavioral
    - Purpose: describe ways in which classes or objects interact with each other
    - Eleven behavioral patterns

GoF Sample Creational Design Patterns

- Singleton
  - Allows only one object instance to be created for the class
  - Provides a global (i.e., public) method that provides access to this one object
- Abstract Factory
  - Provides an interface for creating object instances where objects are related without needing to specify their concrete classes
- Factory Method
  - Provides an interface for creating an object where subclasses decide which class to instantiate

GoF Sample Structural Design Patterns

- Facade
  - Provides a unified interface for a bunch of interfaces within a subsystem

GoF Sample Behavioral Design Patterns

- Command
  - Encapsulates a request as an object
- Iterator
  - Allows sequential access to elements (i.e., objects) within a container (or aggregate) object

What does a software design pattern look like?

- GoF developed a template to describe a design pattern
  - Intent
  - Motivation
  - Applicability
  - Structure
  - Participants
  - Collaborations
  - Consequences
  - Implementation
  - Sample code
  - Known uses
  - Related patterns

GoF Singleton Design Pattern

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it
- Motivation
  - It 's important that some classes only have exactly one instance.
  - E.g.,
    - Only one file system or window manager provided by an OS
  - Make the class itself responsible for keeping track of its sole instance
- Applicability
  - Use Singleton pattern when:
    - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
    - When sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

GoF Singleton Design Pattern

```
Singleton
-instance: static Singleton
-otherAttributes
#Singleton()
+getInstance(): static Singleton
+otherMethods()
```

- Structure
  - i.e., see class diagram
- Participants
  - Singleton
    - Defines a protected constructor and a public static getInstance() operation
    - Defines a private attribute of type Singleton
- Collaborations
  - Client classes access the instance through the public getInstance() operation
- Consequences
  - Controlled access to sole instance
  - Reduced name space
  - Permits refinement of operations and representation
  - Permits a variable number of instances
    - Pattern can easily be modified if more than one instance becomes necessary

GoF Singleton Design Pattern

```
Singleton
-instance: static Singleton
-otherAttributes
#Singleton()
+getInstance(): static Singleton
+otherMethods()
```

- Implementation
- Sample code

```
public class Singleton
{
   private static Singleton instance = null;
   //class may have other attributes!
   protected Singleton()
   {
      //initialize other attributes, if needed
   }
   public static Singleton getInstance()
   {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }
   //class may have other operations!
}
```

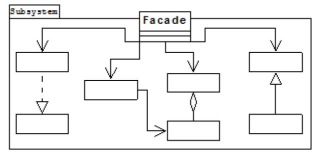
GoF Singleton Design Pattern

- Known uses
  - E.g.,
    - When access to a database needs to be managed by a single database connection
- Related patterns
  - Many patterns can be implemented using the Singleton pattern
  - See Abstract Factory, Builder, and Prototype.

GoF Facade Design Pattern

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem
  - Facade defines a higher-level interface that makes the subsystem easier to use
- Motivation
  - Structuring a system into subsystems helps reduce complexity
  - Want to minimize communication and dependencies between subsystems
- Applicability
  - Use Facade pattern when:
    - You want to provide a simple interface to a complex subsystem
    - There are many dependencies between clients and the subsystem. Introduce a facade to decouple the clients from the subsystem

GoF Facade Design Pattern



- Structure
  - i.e., see class diagram
- Participants
  - Facade
    - Knows which subsystem classes are responsible for a request
    - Delegates client requests to appropriate subsystem objects
  - Subsystem classes
    - Implement subsystem functionality; handle work assigned by Facade object
    - Have no knowledge of Facade i.e., they have no references to it
- Collaborations
  - Clients communicate with subsystem by sending requests to Facade
  - Facade forwards each request to appropriate subsystem object
  - Clients that use Facade do not have access to subsystem objects
- Consequences
  - Shields clients from subsystem complexity; promotes weak coupling
  - Does not prevent applications from directly using subsystem objects

GoF Facade Design Pattern

- Implementation
- Sample code

```
public class Compiler    //Facade to a compiler subsystem
{
   private Scanner scan;
   private Parser parser;                    //compiler subsystem class
   private ProgramNodeBuilder builder;       //compiler subsystem class
   public Compiler()
   {
      buoilder = new ProgramNodeBuilder(...);
      parser = new Parser(...);
   }
   public void compile(File codeFile, BytecodeStream output)
   {
      scan = new Scanner(codeFile);
      parser.parse(scan, builder);
      CodeGenerator generator = new CodeGenerator(builder);
      generator.generate(output);
   }
}
```

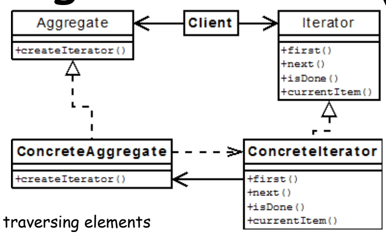
GoF Facade Design Pattern

- Known uses
  - E.g.,
    - When access to a complex subsystem needs to be simplified
- Related patterns
  - Abstract Factory may be used by Facade to provide an interface for creating subsystem objects
  - Mediator is similar to Facade
    - But Mediator's purpose is to abstract arbitrary communication between colleague objects
    - Mediator may centralize functionality that does not belong in any of the colleague classes
    - Mediator's colleagues are aware of and communicate with Mediator object (instead of communicating with each other directly)
  - Facade objects are often Singletons

GoF Iterator Design Pattern

- Intent
  - Provide way to access elements of an aggregate object sequentially without exposing its underlying representation
- Motivation
  - An aggregate object should give you a way to access its elements without exposing its internal structure
  - You may want to traverse the aggregate object in different ways, but you do not want to make the interface larger by adding operations for different traversals
- Applicability
  - Use Iterator pattern to:
    - Access an aggregate object's contents without exposing its internal representation
    - Support multiple traversals of aggregate objects
    - Provide a uniform interface for traversing different aggregate structures

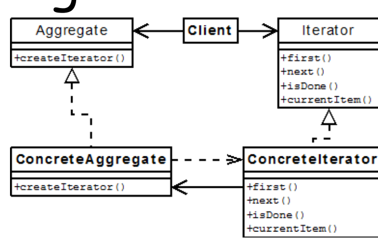
GoF Iterator Design Pattern



- Structure
  - i.e., see class diagram
- Participants
  - Iterator
    - Defines an interface for accessing and traversing elements
  - ConcreteIterator
    - Implements Iterator interface; keeps track of current position in traversal of aggregate
  - Aggregate
    - Defines an interface for creating an Iterator object
  - ConcreteAggregate
    - Implements Iterator creation interface to return an instance of the proper ConcreteIterator

CSC 276 Object-oriented Software Design Slide 19

GoF Iterator Design Pattern



- Collaborations
  - A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal
- Consequences
  - Supports variations in the traversal of an aggregate
  - Simplifies the aggregate interface
  - More than one traversal may be pending on an aggregate

CSC 276 Object-oriented Software Design Slide 20

GoF Iterator Design Pattern

- Implementation
- Sample code
  - Not provided
- Known uses
  - Common in OO systems; most collections (i.e., data structures) provide one or more iterators
- Related patterns
  - Iterators are often applied to recursive structures e.g., Composite

CSC 276 Object-oriented Software Design Slide 21

Larman Sample Design Patterns

- Defines nine patterns
- Characterized as **general responsibility assignment software patterns** (GRASP)

CSC 276 Object-oriented Software Design Slide 22

Larman Sample Design Patterns (cont'd)

- Creator
  - Similar to GoF Factory patterns
  - Describes a design solution where a class is responsible for creating object instances of another class
- Low Coupling
  - Assigns a responsibility in a way that decreases coupling between classes
- High Cohesion
  - Assigns a responsibility in a way that increases cohesion within a class

What does a software design pattern look like?

- Larman developed a template to describe a design pattern
  - Problem
  - Solution
  - Discussion
  - Contraindications
  - Benefits
  - Related patterns or principles

Larman Creator Design Pattern

- Problem
  - Who should be responsible for creating a new instance of some class?
  - Consequence?
    - Creating objects is a common activity in OO systems
- Solution
  - Assign class B responsibility to create an instance of class A if any of the following are true
    - B contains A
    - B aggregates A
    - B has the initializing data for A
    - B records A
    - B closely uses A

Larman Creator Design Pattern (cont'd)

- Discussion
  - Look for class that needs connection to created object
  - Look for common relationships
    - Aggregate *aggregates* Part
    - Container *contains* Content
    - Recorder *records* Recorded
- Contraindications
  - When creation involves significant complexity, use Factory
- Benefits
  - Lowers coupling, which implies lower maintenance costs
- Related patterns or principles
  - Low Coupling, Factory, Whole-Part

Larman Low Coupling Design Pattern

- Problem
  - How to support low dependency, low change impact, and increased reuse?
  - A class with high coupling relies on many other classes
  - This is undesirable since:
    - Changes in related classes force local changes
    - Harder to understand in isolation
    - Harder to reuse since its use requires additional classes
- Solution
  - Assign a responsibility so that coupling remains low

Larman Low Coupling Design Pattern (cont'd)

- Discussion
  - Common forms of coupling:
    - Class Q has an attribute of type X
    - A Q object calls operations using an X object
    - Class Q has a method that refers to an X object
    - Class X is a subclass of Q
    - X is an interface and Q implements it
- Contraindications
  - High coupling to a widely used library (e.g, Java API) is okay
- Benefits
  - High coupling by itself may not be a problem
  - It's high coupling to elements that are unstable
    - E.g., interface changes frequently, implementation changes frequently

Larman High Cohesion Design Pattern

- Problem
  - How to keep complexity manageable?
  - A class with low cohesion does many unrelated things
  - This is undesirable since class is:
    - Hard to comprehend
    - Hard to reuse
    - Hard to maintain
    - Delicate; constantly affected by change
- Solution
  - Assign a responsibility so that cohesion remains high

CSC 276 Object-oriented Software Design Slide 29

Larman High Cohesion Design Pattern (cont'd)

- Discussion
  - Grady Booch
    - High cohesion exists when elements of a component (e.g., a class) "all work together to provide some well-bounded behavior"
  - Rule of thumb
    - A highly cohesive class:
      - Has a relatively small number of methods with highly related functionality
      - Does not do too much work
      - Collaborates with other objects to share effort when task is too large
- Contraindications
  - Improved performance may be a reason to design a class with lower cohesion
- Benefits
  - Clarity and ease of understanding the design
  - Maintenance/enhancements are simplified
  - Low coupling is often a by-product
  - Improve s reuse of classes

CSC 276 Object-oriented Software Design Slide 30

Fernandez Sample Security Design Patterns

- Defines security patterns that describe design solutions to various security concerns
- Categorized using a matrix that shows multiple dimensions
- Dimensions include
  - Life cycle phases
    - e.g., domain analysis, design
  - Levels of architecture
    - e.g., application, operating system, distribution, transport, network
  - Purpose
    - e.g., filtering, access control, authentication

Fernandez Sample Security Design Patterns (cont'd)

- Symmetric Encryption
  - Describes use of encryption to make a message unreadable unless you have the key
  - Same key is used to encrypt and decrypt message
- Asymmetric Encryption
  - Describes use of encryption to make a message unreadable unless you have the key
  - A public key used to encrypt message
  - A private key used to decrypt message

Fernandez Sample Security Design Patterns (cont'd)

- Digital Signature with Hashing
  - Describes a way to allow sender of a message to prove that the message was originated from them and not someone else
  - Also describes how receiver of message can verify the integrity of the message
    - i.e., that it has not been altered during transmission
- Secure Three-Tier Architecture
  - Describes model-view-controller components as tiers (or layers) of a distributed system
  - Each tier enforces security applicable to the tier

What does a software design pattern look like?

- Fernandez developed a template to describe a design pattern
  - Example
  - Context
  - Problem
  - Solution
  - Implementation
  - Example resolved
  - Consequences
  - Known uses
  - See also

Fernandez Symmetric Encryption Design Pattern

- Example
  - Alice sends sensitive data to Bob
  - Eve can intercept this data; reads sensitive data
- Context
  - Applications exchange sensitive information over insecure channels
- Problem
  - Sensitive data may be read by unauthorized users while in transit (or at rest)
- Solution
  - Sender: transform plaintext data into ciphertext using a secret key
  - Sender: transmit ciphertext over the insecure channel
  - Receiver: transform ciphertext into plaintext data using same secret key

CSC 276 Object-oriented Software Design Slide 35

Fernandez Symmetric Encryption Design Pattern
(cont'd)

- Implementation
  - Both sender and receiver need to agree on
    - Cryptography algorithm
    - Secret key
- Example resolved
  - Alice encrypts sensitive data then sends ciphertext to Bob
  - Eve can still intercept this data; but cannot read sensitive data
- Consequences
  - Key needs to be secret; need to share in secure way
  - Selection of crypto algorithm and key length impacts performance and level of security

CSC 276 Object-oriented Software Design Slide 36

Fernandez Symmetric Encryption Design Pattern
(cont'd)

- Known uses
  - GNuPG
  - OpenSSL
  - Java Cryptographic Extension
  - .NET framework
  - XML encryption
  - Pretty Good Encryption (PGP)
- See also
  - Secure channel communication pattern
  - Asymmetric encryption
  - Patterns for key management

Fernandez Asymmetric Encryption Design Pattern

- Example
  - Alice needs to send sensitive data to Bob, but they do not share a secret key
  - Eve can intercept this data; reads sensitive data
- Context
  - Applications exchange sensitive information over insecure channels
- Problem
  - Sensitive data may be read by unauthorized users while in transit (or at rest)
- Solution
  - Sender: transform plaintext data into ciphertext using receiver's public key
  - Sender: transmit ciphertext over the insecure channel
  - Receiver: transform ciphertext into plaintext data using their private key

Fernandez Asymmetric Encryption Design Pattern
(cont'd)

- Implementation
  - Use well-known algorithm (e.g., RSA)
  - Both sender and receiver need to agree on cryptography algorithm
- Example resolved
  - Alice looks up Bob's public key, uses to encrypt sensitive data then sends ciphertext to Bob
  - Eve can still intercept this data; but cannot read sensitive data
  - Bob can decrypt using his private key
- Consequences
  - Anyone can look up someone's public key
  - Selection of crypto algorithm and key length impacts performance and level of security

Fernandez Asymmetric Encryption Design Pattern
(cont'd)

- Known uses
  - GNuPG
  - Java Cryptographic Extension
  - .NET framework
  - XML encryption
  - Pretty Good Encryption (PGP)
- See also
  - Secure channel communication pattern

Fernandez Digital Signature with Hashing Design Pattern

- Example
  - Alice wants to send non-sensitive data to Bob
  - Bob wants to make sure data came from Alice
  - Eve can intercept this data and modify it
- Context
  - Applications exchange information over insecure channels
  - Application may need to confirm integrity and origin of the data
- Problem
  - Need to authenticate the origin of the message (data)

Fernandez Digital Signature with Hashing Design Pattern

- Solution
  - Sender: compute digest on plaintext data using a hash function
  - Sender: transform plaintext data into ciphertext using sender's private key
  - Sender: send both digest and ciphertext
  - Receiver: decrypt ciphertext using sender's public key
  - Receiver: compute digest on decrypted ciphertext
  - Receiver: compare its computed digest with digest received from sender
- Implementation
  - Use a cryptographic hash function; these are better at producing unique digests that are hard to reverse into the original plaintext
  - Sender and receiver must agree on hash function and asymmetric cryptographic algorithm

Fernandez Digital Signature with Hashing Design Pattern

- Example resolved
  - Alice now uses an asymmetric algorithm and a hash function to send non-sensitive data to Bob
  - Bob verifies that his computed digest matches what Alice sent him
  - Eve can intercept this data, but cannot decrypt the data or use the hash digest
- Consequences
  - Sender cannot deny that they sent the message (assuming their private key is only known by them)
- Known uses
  - GNuPG
  - Java Cryptographic Architecture
  - .NET framework
  - XML signature

Fernandez Secure Three-Tier Architecture Pattern

- Context
  - Applicable to distributed systems in homogeneous or heterogeneous environments
- Problem
  - Need to secure all tiers of system; having an insecure tier/layer invites attacks
  - Attacks may come from legitimate users
  - Provide services that are available through mostly transparent security features
  - Be able to show that a user performed an action; that the user cannot deny an action they performed

Fernandez Secure Three-Tier Architecture Pattern

- Solution
  - Apply appropriate security services to each layer/tier
    - E.g., use encryption on data sent between the layers
  - Presentation layer
    - Require authentication and authorization of users
  - Business layer
    - Define a unified access control model
  - Storage layer
    - Consider encrypting sensitive data
- Implementation
  - Define global authorization model
  - Select authentication approaches based on needs of applications
  - Select an encryption approach

Fernandez Secure Three-Tier Architecture Pattern

- Consequences
  - Centralized security
    - Authorization constraints, authentication information and logging repositories
  - All layers apply security restrictions
    - Security is transparent to user (if possible)
  - Availability
  - Non-repudiation
  - Consider security overhead
- Known uses
  - Web services, distributed apps

Summary: Software Design Pattern Templates

GoF	Larman	Fernandez
Original	GRASP	Security
Intent Motivation Applicability Structure Participants Collaborations Consequences Implementation Sample code Known uses Related patterns	Problem Solution Discussion Contraindications Benefits Related patterns/principles	Example Context Problem Solution Implementation Example resolved Consequences Known uses See also