Innehåll
Vi lånar två av fyra uppsättningar av Bruegges OH-bilder till kapitel 6
System Design Design Patterns s 2-42
System Design Design Patterns II s 43-109
Techniques for Finding Objects

- Requirements Analysis
  - Start with Use Cases. Identify participating objects
  - Textual analysis of flow of events (find nouns, verbs, ...)
  - Extract application domain objects by interviewing client (application domain knowledge)
  - Find objects by using general knowledge
- System Design
  - Subsystem decomposition
  - Try to identify layers and partitions
- Object Design
  - Find additional objects by applying implementation domain knowledge

Another Source for Finding Objects: Design Patterns

- Observation [Gamma et al 95]:
  - Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's.
  - There is a need for reusable and flexible designs
  - Design knowledge complements application domain knowledge and implementation domain knowledge.
  - What are Design Patterns?
    - A design 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 the this solution a million times over, without ever doing it the same twice

Design Patterns Notation

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995
- Based on OMT Notation
- Notational issues
  - Attributes come after the Operations
  - Associations are called acquaintances
  - Multiplicities are shown as solid circles
  - Inheritance shown as triangle
  - Dashed line: Instantiation Association (Class can instantiate objects of associated class) (In UML it denotes a dependency
  - UML Note is called Dogear box (connected by dashed line to class operation): Pseudo-code implementation of operation

Review: Modeling Typical Aggregations

Fixed Structure:

Organization Chart (variable aggregate):

Dynamic tree (recursive aggregate):

Compound Statement

Simple Statement
Review: Modeling Typical Aggregations

Fixed Structure:
- Car
  - Doors
  - Wheels
  - Battery
  - Engine

Organization Chart (variable aggregate):
- University
  - School
    - Department

Composite Pattern

Graphical Applications use Composite Patterns

- The Graphic Class represents both primitives (Line, Circle) and their containers (Picture)

Modeling Software Development with Composite Patterns

- Software Lifecycle:
  - Definition: The software lifecycle consists of a set of development activities which are either other activities or collection of tasks
  - Composite: Activity (The software lifecycle consists of activities which consist of activities, which consist of activities, which...)
  - Leaf node: Task
- Software System:
  - Definition: A software system consists of subsystems which are either other subsystems or collection of classes
  - Composite: Subsystem (A software system consists of subsystems which consists of subsystems, which consists of subsystems, which...)
  - Leaf node: Class

Composite Pattern

- Composes objects into tree structures to represent part-whole hierarchies with arbitrary depth and width.
- The Composite Pattern lets clients treat individual objects and compositions of these objects uniformly

Client

Component

Leaf

Operation()
Modeling the Software Lifecycle with a Composite Pattern

Manager

Software Lifecycle

Task

Activity

Children

Modeling a Software System with a Composite Pattern

Developer

Software System

Class

Subsystem

Children

Ideal Structure of a Subsystem: Façade, Adapter, Bridge

- A subsystem consists of
  - an interface object
  - a set of application domain objects (entity objects) modeling real entities or existing systems
    - Some of the application domain objects are interfaces to existing systems
  - one or more control objects
- Realization of Interface Object: Façade
  - Provides the interface to the subsystem
- Interface to existing systems: Adapter or Bridge
  - Provides the interface to existing system (legacy system)
  - The existing system is not necessarily object-oriented!

Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e., it abstracts out the gory details)
- Facades allow us to provide a closed architecture
Open vs Closed Architecture

- Open architecture:
  - Any client can see into the vehicle subsystem and call on any component or class operation at will.
  - Why is this good?
    - Efficiency
  - Why is this bad?
    - Can’t expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
    - We can be assured that the subsystem will be misused, leading to non-portable code.

Realizing a Closed Architecture with a Facade

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers.
- If a façade is used the subsystem can be used in an early integration test.
- We need to write only a driver.

Realizing a Compiler with a Facade pattern

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers.
- If a façade is used the subsystem can be used in an early integration test.
- We need to write only a driver.

UML Notation for subsystems: Package

- Package = Collection of classes that are grouped together
- Packages are often used to model subsystems
- Notation:
  - A box with a tab.
  - The tab contains the name of the package
- In Together-J, every class is assigned to a default package.
  - When you create a class, the class is assigned to the default package directly containing the class diagram.
  - You can create other packages, but cannot delete the default package.
Class Adapter Pattern
(based on Multiple Inheritance)

Client

Target
Request()

Adaptee (Legacy Object)
ExistingRequest()

Adapter
Request()
Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation.
- Example: I have a List class, I need a Stack class. How about subclassing the Stack class from the List class and providing three methods, Push() and Pop(), Top()?
- Problem with implementation inheritance:
  Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?

Problem with implementation inheritance:

Delegation

- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
  - A receiving object delegates operations to its delegate.
  - The developer can make sure that the receiving object does not allow the client to misuse the delegate object

Delegation or Inheritance?

- Delegation
  - Pro:
    - Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)
  - Con:
    - Inefficiency: Objects are encapsulated.
- Inheritance
  - Pro:
    - Straightforward to use
    - Supported by many programming languages
    - Easy to implement new functionality
  - Con:
    - Inheritance exposes a subclass to the details of its parent class
    - Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)

Delegation instead of Inheritance

- Delegation: Catching an operation and sending it to another object.
Many design patterns use a combination of inheritance and delegation

**Adapter Pattern**

- “Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces
  - Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
  - Also known as a wrapper
  - Two adapter patterns:
    - Class adapter: Uses multiple inheritance to adapt one interface to another
    - Object adapter: Uses single inheritance and delegation
  - We will mostly use object adapters and call them simply adapters

**Adapter pattern**

- Delegation is used to bind an Adapter and an Adaptee
- Interface inheritance is used to specify the interface of the Adapter class.
- Target and Adaptee (usually called legacy system) pre-exist the Adapter.
- Target may be realized as an interface in Java.

**Adapter pattern example**

```java
public class ServicesEnumeration implements Enumeration {
    private int currentServiceIdx;
    private int numServices();

    public boolean hasMoreElements() {
        return this.currentServiceIdx < numServices();
    }
    public Object nextElement() {
        if (!this.hasMoreElements())
            throw new NoSuchElementException();
        return adaptee.getService(this.currentServiceIdx++);
    }
}
```
Bridge Pattern

- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”. (From [Gamma et al 1995])
- Also know as a Handle/Body pattern.
- Allows different implementations of an interface to be decided upon dynamically.

Using a Bridge

- The bridge pattern is used to provide multiple implementations under the same interface.
- Examples: Interface to a component that is incomplete, not yet known or unavailable during testing
- JAMES Project (WS 97-98): if seat data is required to be read, but the seat is not yet implemented, not yet known or only available by a simulation, provide a bridge:

\[
\begin{align*}
\text{VIP} & \quad \text{Seat} \quad \text{impl} \\
\text{GetPosition()} & \quad \text{setPosition()} \\
\text{Stub Code} & \quad \text{AIMSeat} \quad \text{SARTSeat}
\end{align*}
\]

JAMES Bridge Example

```java
public interface SeatImplementation {
    public int GetPosition();
    public void setPosition(int newPosition);
}

public class AimSeat implements SeatImplementation {
    public int GetPosition() {
        // actual call to the AIM simulation system
    }
    ...
}

public class SARTSeat implements SeatImplementation {
    public int GetPosition() {
        // actual call to the SART seat simulator
    }
    ...
}
```

Bridge Pattern

[Diagram of Bridge Pattern]

[Diagram of JAMES Bridge Example]
**Adapter vs Bridge**

- **Similarities:**
  - Both used to hide the details of the underlying implementation.
- **Difference:**
  - The adapter pattern is geared towards making unrelated components work together
    - Applied to systems after they’re designed (reengineering, interface engineering).
  - A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.
  - Green field engineering of an “extensible system”
  - New “beasts” can be added to the “object zoo”, even if these are not known at analysis or system design time.

**Example for Combination of Adapters and Bridges in JAMES**

- **Seat Preferences**
  - GetSeatPos()
  - SetSeatPos()

- **Seat Impl**
  - Adapter

- **Bridge**
  - SLBRDR32
  - Existing Smart Card Library from Schlumberger
  - Seats for the Car

- **AIMSeat**
- **SARTSeat**
- **ActualSeat**
- **PreferencesCardlet**

---

**Design Patterns encourage good Design Practice**

- A facade pattern should be used by all subsystems in a software system. The façade defines all the services of the subsystem.
  - The façade will delegate requests to the appropriate components within the subsystem.
- Adapters should be used to interface to any existing proprietary components.
  - For example, a smart card software system should provide an adapter for a particular smart card reader and other hardware that it controls and queries.
- Bridges should be used to interface to a set of objects where the full set is not completely known at analysis or design time.
  - Bridges should be used when the subsystem must be extended later (extensibility).

---

**Other Design Heuristics**

- Never use implementation inheritance, always use interface inheritance
- A subclass should never hide operations implemented in a superclass
- If you are tempted to use implementation inheritance, use delegation instead
**Summary**

- **Composite Pattern:**
  - Models trees with dynamic width and dynamic depth
- **Facade Pattern:**
  - Interface to a Subsystem
  - Closed vs Open Architecture
- **Adapter Pattern:**
  - Interface to Reality
- **Bridge Pattern:**
  - Interface Reality and Future
- Read and Reread Design Patterns Book
  - Learn how to use it as a reference book

**Patterns covered in next lecture**

- **Creational Patterns**
  - Abstract Factory Pattern ("Device Independence")
- **Structural Patterns**
  - Proxy ("Location Transparency")
- **Behavioral Patterns**
  - Command ("Request Encapsulation", "unlimited undos")
  - Observer ("Publish and Subscribe")
  - Strategy ("Policy vs Mechanism", "Encapsulate family of algorithms")

**Outline**

- A little discourse on Advanced Object modeling
  - Sequence Diagrams:
  - Tie together use cases and participating objects
  - Object types: Entity objects, boundary objects, control objects
- Review of design pattern concepts
  - What is a design pattern?
  - Modifiable designs
- More patterns:
  - Abstract Factory
  - Proxy
  - Command
  - Observer
  - Strategy
**Sequence Diagrams**

- Use cases and participating objects are found. What now?
  - Sequence diagram - A diagram that shows object interactions arranged in time sequence for a specific use case or scenario.
  - A sequence diagram includes time but does not include object relationships.
  - Sequence diagrams are used to describe use cases (i.e., all possible interactions between participating objects) and scenarios (i.e., one possible interaction, see next slide)
  - In other words: A sequence diagram is useful to model a use case or scenario with its participating objects. It often leads to the detection of new participating objects.

**Drawing Sequence Diagrams**

- Each column represents an object that is participating in the interaction.
- The vertical axis represents time (from top to bottom). Messages are shown by full arrows.
- Labels on full arrows represent message names and arguments.
- Activations (i.e., the time it takes to perform an operation) are depicted by a rectangle attached to an object. The height of the rectangle is indicative for the duration of the operation
  - The vertical rectangle shows an object is active, that is, it is handling a request made by another object.
  - The operation can itself send other requests to other objects
  - An object can request an operation from itself (looping arrow)

**Sequence Diagrams vs Interaction Diagrams**

- In the Design Pattern book sequence diagrams are called interaction diagrams.
  - A solid vertical line indicates the lifetime of a particular object
  - If the object does not get instantiated until after the beginning of time as recorded in the diagram, then its vertical line is dashed until the point of creation, where it becomes solid.
Increase the Reuse of your Model even more:
Structuring the Participating Objects

Different Object Types

- Boundary objects
  - Implement the interaction with the user
  - Constructed from UI components
  - Subject to most modification
- Entity objects
  - Represent the domain model
  - Often represent persistent data
  - Least often modified
- Control objects
  - Implement the transactions with the user
  - Constructed with Command objects
  - Modified frequently but less often than boundary objects

Identifying Boundary Objects

- Boundary objects model the system interface with the actors.
- In each use case, each actor interacts at least through one boundary object.
- The boundary object collects the information from the actor and translates it into an interface neutral form that can be used by the control and entity objects.
Identifying Control Objects

- Control objects are responsible for coordinating boundary and entity objects.
- There is often a close relationship between a use case and a control object.
- A control object is usually created at the beginning of a use case and ceases to exist at its end.
- It is responsible for collecting information from the boundary objects and dispatching it to entity objects.
- For example, control objects describe the behavior associated with the sequencing of forms, undo and history queues, and dispatching information in a distributed system.

Some Heuristics for Identification of Interface and Control Objects

- Identify one control object per use case or more if the use case is complex and can be divided into shorter flows of events.
- Identify one control object per actor in the use case.
- The life span of a control object should be extent of the use case or the extent of a user session. If it is difficult to identify the beginning and the end of a control object activation, the corresponding use case may not have a well-defined entry and exit condition.

Some Heuristics for Good Sequence Diagrams

- Column 1 models the actor who initiated the use case.
- Column 2 should be a boundary object (that the actor used to initiate the use case).
- Column 3 should be the control object that manages the rest of the use case.
- Control objects are created by boundary objects initiating use cases.
- Boundary objects are created by control objects.
- Entity objects are accessed by control and boundary objects.
- Entity objects never access boundary or control objects. This makes it easier to share them across use cases.

Back to the main topic: Design patterns

A design pattern is...

...a template solution to a recurring design problem
  * Look before re-inventing the wheel just one more time

...reusable design knowledge
  * Higher level than classes or data structures (link lists, binary trees...)
  * Lower level than application frameworks

...an example of modifiable design
  * Learning to design starts by studying other designs
Why are modifiable designs important?

A modifiable design enables...

...an iterative and incremental development cycle
  - concurrent development
  - risk management
  - flexibility to change

...to minimize the introduction of new problems when fixing old ones

...to deliver more functionality after initial delivery

What makes a design modifiable?

- Low coupling and high coherence
- Clear dependencies
- Explicit assumptions

How do design patterns help?

- They are generalized from existing systems
- They provide a shared vocabulary to designers
- They provide examples of modifiable designs
  - Abstract classes
  - Delegation
Interface and class inheritance in Java

- An interface defines constants and abstract methods.
  - An interface can extend (i.e., inherit from) zero, one, or many interfaces.
  - An interface does not define any behavior or attributes.

- A class defines attributes and methods.
  - A class can extend (i.e., inherit behavior and attributes from) zero or one class.
  - A class can implement (i.e., comply with) zero, one, or many interfaces in addition to extending a super class.

- Abstract classes should be realized by interfaces in Java.
- Concrete classes should be realized by classes in Java.

Proxy Pattern: Motivation

- It is 15:00pm. I am sitting at my 14.4 baud modem connection and retrieve a fancy web site from the US. This is prime web time all over the US. So I am getting 10 bits/sec.
- What can you do?

Proxy Pattern

- What is expensive?
  - Object Creation
  - Object Initialization
- Defer object creation and object initialization to the time you need the object
- Proxy pattern:
  - Reduces the cost of accessing objects
  - Uses another object ("the proxy") that acts as a stand-in for the real object
  - The proxy creates the real object only if the user asks for it

On to More Patterns!

- Structural pattern
  - Proxy
- Creational Patterns
  - Abstract Factory
  - Builder
- Behavioral pattern
  - Command
  - Observer
  - Strategy
Proxy pattern (207)

- Interface inheritance is used to specify the interface shared by Proxy and RealSubject.
- Delegation is used to catch and forward any accesses to the RealSubject (if desired)
- Proxy patterns can be used for lazy evaluation and for remote invocation.
- Proxy patterns can be implemented with a Java interface.

Proxy Applicability

- Remote Proxy
  - Local representative for an object in a different address space
  - Caching of information: Good if information does not change too often (PAID!)
- Virtual Proxy
  - Object is too expensive to create or too expensive to download
  - Proxy is a stand-in
- Protection Proxy
  - Proxy provides access control to the real object
  - Useful when different objects should have different access and viewing rights for the same document.
  - Example: Grade information for a student shared by administrators, teachers and students.

Virtual Proxy example

- Images are stored and loaded separately from text
- If a RealImage is not loaded a ProxyImage displays a grey rectangle in place of the image
- The client cannot tell that it is dealing with a ProxyImage instead of a RealImage
- A proxy pattern can be easily combined with a Bridge

Before
Towards a Pattern Taxonomy

- **Structural Patterns**
  - Adapters, Bridges, Facades, and Proxies are variations on a single theme:
    - They reduce the coupling between two or more classes
    - They introduce an abstract class to enable future extensions
    - Encapsulate complex structures

- **Behavioral Patterns**
  - Concerned with algorithms and the assignment of responsibilities between objects: Who does what?
  - Characterize complex control flow that is difficult to follow at runtime.

- **Creational Patterns**
  - Abstract the instantiation process.
  - Make a system independent from the way its objects are created, composed and represented.
Command Pattern: Motivation

- You want to build a user interface
- You want to provide menus
- You want to make the user interface reusable across many applications
  - You cannot hardcode the meanings of the menus for the various applications
  - The applications only know what has to be done when a menu is selected.
- Such a menu can easily be implemented with the Command Pattern

Command pattern

- **Client** creates a **ConcreteCommand** and binds it with a **Receiver**.
- **Client** hands the **ConcreteCommand** over to the **Invoker** which stores it.
- The **Invoker** has the responsibility to do the command ("execute" or "undo").

Menu Example

```
Document
  action()
  *  binds
  ConcreteCommand
```

Command pattern Applicability

- "Encapsulate a request as an object, thereby letting you
  - parameterize clients with different requests,
  - queue or log requests, and
  - support undoable operations."

- Uses:
  - Undo queues
  - Database transaction buffering
Command pattern: Editor with unlimited undos

Structuring the objects

A Pattern Taxonomy
Observer pattern (293)

- “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.” (p. 293)
- Also called “Publish and Subscribe”

- Uses:
  - Maintaining consistency across redundant state
  - Optimizing batch changes to maintain consistency

Observer pattern (continued)

- The Subject represents the actual state, the Observers represent different views of the state.
- Observer can be implemented as a Java interface.
- Subject is a super class (needs to store the observers vector) not an interface.

Sequence diagram for scenario:
Change filename to “foo”

Subject goes through all its observers and calls update() on them, asking for the new state is decoupled from the notification.
Example from a Real Project: JAMES Project

- Nonfunctional Requirements
  - Foundation for wide range of smartcard applications
  - Bonus system, vehicle personalization, travel planning,
  - Flexibility and extendability for future applications
  - Integration of various computing platforms and devices
  - Central database hosts, home and dealership PCs, onboard computer, speech system, device controllers, smartcards
  - Scalability
  - Support for large number of mobile clients and servers
  - Openness
  - Use of open standards and commercial components

These requirements lead to the use of the observer pattern -> JAMES Event Service

James Event Service

Event channels are the main means of communication in James
currently, only used for onboard communication between various subsystems in vehicle

however, could be used also for wide-area communication
**JAMES Event Service Requirements**

- Peers can be created and deleted dynamically.
- Peers can subscribe dynamically as publishers or subscribers.
- A peer can subscribe to many event channels at the same time.
- A peer can be publisher and subscriber on an event channel at the same time.
- Publishers do not know their subscribers (only the event channel).
- Event channels allow plug-and-play with cardlets.

Use of the Observer Pattern leads to scalable and extensible architecture.

**James Event Service as a set of objects**

- View ("Boundary objects"): Viewlets
- Controller ("Control Object"): Event Channel
- Model ("Entity Objects"): Cardlets and Vehicle Subsystem

Cardlets and Viewlets act as peers on the onboard software bus:

- Cardlets are applications running on a Java smartcard.
- Viewlets are the corresponding GUI frontends on the onboard computer.

Cardlet Communication in James:

[Diagram showing cardlet communication in the James system.]
Strategy Pattern

- Many different algorithms exist for the same task
- Examples:
  - Breaking a stream of text into lines
  - Parsing a set of tokens into an abstract syntax tree
  - Sorting a list of customers
- The different algorithms will be appropriate at different times
  - Rapid prototyping vs delivery of final product
- We don’t want to support all the algorithms if we don’t need them
- If we need a new algorithm, we want to add it easily without disturbing the application using the algorithm
**Applicability of Strategy Pattern**

- Many related classes differ only in their behavior. Strategy allows to configure a single class with one of many behaviors.
- Different variants of an algorithm are needed that trade-off space against time. All these variants can be implemented as a class hierarchy of algorithms.

**Abstract Factory Motivation**

- Consider a user interface toolkit that supports multiple looks and feel standards such as Motif, Windows 95 or the finder in MacOS.
  - How can you write a single user interface and make it portable across the different look and feel standards for these window managers?
- Consider a facility management system for an intelligent house that supports different control systems such as Siemens’ Instabus, Johnson & Control Metasys or Zumtobe’s proprietary standard.
  - How can you write a single control system that is independent from the manufacturer?

**A Pattern Taxonomy**

**Abstract Factory**

![Diagram of Abstract Factory pattern with classes and objects]
Applicability for Abstract Factory Pattern

- Independence from Initialization or Representation:
  - The system should be independent of how its products are created, composed or represented.
- Manufacturer Independence:
  - A system should be configured with one of multiple family of products.
  - You want to provide a class library for a customer ("facility management library"), but you don’t want to reveal what particular product you are using.
- Constraints on related products:
  - A family of related products is designed to be used together and you need to enforce this constraint.
- Cope with upcoming change:
  - You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.

Builder Pattern Motivation

- Conversion of documents:
- Software companies make their money by introducing new formats, forcing users to upgrades.
- But you don’t want to upgrade your software every time there is an update of the format for Word documents.
- Idea: A reader for RTF format.
  - Convert RTF to many text formats (EMACS, Framemaker 4.0, Framemaker 5.0, Framemaker 5.5, HTML, SGML, WordPerfect 3.5, WordPerfect 7.0, ....)
- Problem: The number of conversions is open-ended.
- Solution:
  - Configure the RTF Reader with a “builder” object that specializes in conversions to any known format and can easily be extended to deal with any new format appearing on the market.
### Example

```java
Example

```Parse()
RTFReader

\while i = GetNextToken()
\switch t.Type
\case CHAR: builder->ConvertCharacter(t.Char)
\case FONT: builder->ConvertFont(t.Font)
\case PARA: builder->ConvertParagraph()
\end

```

### ConvertCharacter

```java
ConvertCharacter()
```

### ConvertFont

```java
ConvertFont()
```

### ConvertParagraph

```java
ConvertParagraph()
```

### Applicability for Builder Pattern

- The creation of a complex product must be independent of the particular parts that make up the product
  - In particular, the creation process should not know about the assembly process (how the parts are put together to make up the product)
  - The creation process must allow different representations for the object that is constructed. Examples:
    - A house with one floor, 3 rooms, 2 hallways, 1 garage and three doors.
    - A skyscraper with 50 floors, 15 offices and 5 hallways on each floor. The office layout varies for each floor.

### Abstract Factory vs Builder

- **Abstract Factory**
  - Focuses on product family
  - The products can be simple (“light bulb”) or complex
  - The abstract factory does not hide the creation process
    - The product is immediately returned
  - **Builder**
    - The underlying product needs to be constructed as part of the system but is very complex
    - The construction of the complex product changes from time to time
    - The builder pattern hides the complex creation process from the user:
      - The product is returned after creation as a final step

- **Abstract Factory and Builder work well together for a family of multiple complex products**

### Summary

- **Structural Patterns**
  - Focus: How objects are composed to form larger structures
  - Problems solved:
    - Realize new functionality from old functionality,
    - Provide flexibility and extensibility
- **Behavioral Patterns**
  - Focus: Algorithms and the assignment of responsibilities to objects
  - Problem solved:
    - Too tight coupling to a particular algorithm
- **Creational Patterns**
  - Focus: Creation of complex objects
  - Problems solved:
    - Hide how complex objects are created and put together
Conclusion

- Design patterns
  - Provide solutions to common problems.
  - Lead to extensible models and code.
  - Can be used as is or as examples of interface inheritance and delegation.
  - Apply the same principles to structure and to behavior.

- Design patterns solve all your software engineering problems
- My favorites: Composite, Bridge, Builder and Observer