Design Patterns

Bridge, Builder
The Bridge Design Pattern: Structure

Used when:

- An abstraction should be decoupled from its implementation
- The abstraction and its implementation should be (independently) extensible by subclassing
The Bridge Design Pattern: Example (I)

PathFinder

\[ \text{genPath}(j: \text{Job}): \text{Path} \]

ShortestLength

\[ \text{genPath}(j: \text{Job}): \text{Path} \]

ShortestTime

\[ \text{genPath}(j: \text{Job}): \text{Path} \]
The Bridge Design Pattern: Example (II)

```java
for (Job j : ja)
    ret[i]=genPath(j);
```

- **MultiplePathFinder**
  - genPaths(ja:Job[]):Path[]
  - mergePaths(Path[]):Path

- **PathFinder**
  - genPath(j:Job):Path

- **ShortestLength**
  - genPath(j:Job):Path

- **ShortestTime**
  - genPath(j:Job):Path
The Bridge Design Pattern: Example (III)

```
for (Job j : ja)
    ret[i]=genPath(j);
```

```
PathFinder
    genPath(j:Job):Path

MultiplePathFinder
    genPaths(ja:Job[]):Path[]
    mergePaths(Path[]):Path

ShortestLength
    genPath(j:Job):Path

ShortestTime
    genPath(j:Job):Path

ShortestLengthMLP
    genPath(j:Job):Path

ShortestTimeMLP
    genPath(j:Job):Path
```

→ Spurious subclassing!
The Bridge Design Pattern: Example (IV)

PathFinder
- genPath(j:Job):Path

ShortestLength
- genPath(j:Job):Path

ShortestTime
- genPath(j:Job):Path
The Bridge Design Pattern: Example (V)

PathFinder

-impl

impl.genPathImpl(j);

PathFinderImpl

-genPathImpl(j:Job):Path

ShortestLength

-genPathImpl(j:Job)

ShortestTime

-genPathImpl(j:Job)
The Bridge Design Pattern: Example (VI)

PathFinder

genPath(j:Job):Path

impl.genPathImpl(j);

PathFinderImpl

genPathImpl(j:Job):Path

ShortestLength

genPathImpl(j:Job)

ShortestTime

genPathImpl(j:Job)

MultiplePathFinder

genPaths(ja:Job[]):Path[]
mergePaths(Path[]):Path

for (Job j : ja)
ret[i]=genPath(j);
The Builder Design Pattern: Structure

Used when
- The construction of a product is independent of the parts
- Different representations for the product can be employed
The Builder Design Pattern: Example (I)

Hook Method and Hook Object with the same template method
The Builder Design Pattern: Example (II)

- **Usage:**

```java
SMFloor sparse_floor;
FloorBuilder sm_builder = new SparseMatrixFloor();
LayoutParser layIn = new XMLLayoutParser(sm_builder, xml_file);
layIn.readLayout();
sparse_floor = sm_builder.getSMFloor();
```

- **Change of built object at runtime:**

```java
FMFloor full_floor;
FloorBuilder fm_builder = new FullMatrixFloor();
LayoutParser layIn = new SparseMatrixParser(fm_builder, sparse_floor);
layIn.readLayout();
full_floor = fm_builder.getFMFloor();
```
More Design Patterns

Observer, Command, MVC

Thanks to: A. Naderlinger
The **Observer** pattern

- **Motivation**

- **Intent**
  Define a one-to-many dependency between objects such that when one object changes state, all its dependents are notified and updated automatically.
Observer applicability

- When one aspect of an abstraction depends on another aspect. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

- When a state change in one object requires state changes in others, and you don't know how many objects need to be changed.

- When an object should be able to notify other objects without making assumptions about who these objects are. → In other words, you don't want these objects tightly coupled.
Observer DP structure (UML)
Observer DP structure

```java
for (Observer o : observers) {
    o.update();
}
```
Observer example: event broadcast

Robot

- processEvent(event)
- startMoving()
- stopMoving()
- logErrorMessage()
- saveJobStatus()

RobotState

- powerOff(r:Robot)
- startJob(r:Robot)

state

Ready

- powerOff(r:Robot)
- startJob(r:Robot)

Traveling

- powerOff(r:Robot)
- startJob(r:Robot)

Executing

- powerOff(r:Robot)
- startJob(r:Robot)

Error

- powerOff(r:Robot)
- startJob(r:Robot)

Idle

- powerOff(r:Robot)
- startJob(r:Robot)
Observer example

SystemEventManager
- evtObservers[]: EventObserver
- addEvListener(:EventObserver)
- remEvListener(:EventObserver)
- notifyOnEvent(ev: Event)

EventObserver
- << Interface >>
- processEvent(ev: Event)

Robot
- startUp()
- processEvent(ev: Event)

if (ev.getName.equals(Event.POWEROFF))
    state.poweroff(this)

sysEM
- sysEM.addEvListener(this)
Pull vs. Push

- **Push:**
  - The subject sends detailed information about the change.
  - E.g., all information is encapsulated in an object ("Event")

- **Pull**
  - The subject only informs the observes that there was a change. Observers need to request details themselves.
Interaction

- **Registration**: some component registers as an observer (right at the program initialization or later)
- **Notification**: subject notifies all observers about some change
- **Update**: observer decide on itself how to react on the change. Depending on Push/Pull: observer may fetch information from the subject
- **Deregistration**: if an observer does not want to get further notifications
Observers in Swing: Listeners

```java
//import awt and swing classes
public class SmallSwingEx {

    public static void main(String[] args) {
        JFrame frame = new JFrame("SwingExample");
        final JTextArea someText = new JTextArea();
        JButton bt1 = new JButton("OK");
        JButton bt2 = new JButton("CANCEL");

        ... //add components and define a layout

        bt1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                someText.setText("OK was clicked");
            }
        });

        ...}

        frame.setVisible(true);
    }
}
```
Observer remarks

- Built-in Java classes
- Update order
The **Command** pattern

- **Motivation**
  Issue requests to objects without knowing anything about the operation being requested or the receiver of the request. E.g., UI toolkits

- **Intent**
  Encapsulate a request as an object, thereby letting you parameterize requests, queue or log requests, and support undoable operations.
The **Command** pattern

Why an object?

*Callback function*

- In C/C++: function pointer
- In Java: *Functor* (usually a class with a single method to mimic function pointers)
- In C#: Delegates
The **Command** pattern: structure

```
command.execute()
```

```
Invoker

Command

Receiver

ConcreteCommand

+execute()
```

```
receiver.m()
```
Applicability

- Queue requests, execute them at different times
- Log commands. Reapply them after a system crash
- Compose Macro-Commands (Composite pattern)
- Transactions
- Undo/Redo
Model-View-Controller

- Architectural pattern that combines
  - Observer
  - Strategy
  - Composite
  - Decorator
  - Factory Method
  - ...

- was introduced for building user interfaces in Smalltalk (70s)
MVC stands for:

- **M – model**
  - data and methods to manipulate the data; *business logic*
  - may interact with databases, ...
  - typically consists of multiple classes

- **V – view**
  - visual representation of the model (e.g. GUI)

- **C – controller**
  - takes user input
  - modifies the model
MVC – in principle …

with the Observer pattern at the core
X variants
M-V-C vs. M-VC

- Sometimes V and C are combined (Swing: Model and UI component)

- A dedicated controller class
  - Flexible, extensible design
  - Simplifies View
  - Overcome gap between model-API and user interface
  - Web applications
  - …
Modularization and Software Architectures
Contents

- The notion of software module
- Desired characteristics of modules
- Specification of modules (ADS, ADT)
- Description of software architectures
- Analysis of software architectures
The Software Module
A *module* (software component) is defined as *a piece of software with a programming interface*. One distinguishes between the *interface* of a module and its *implementation*. The possible interaction of several modules is determined by their programming interfaces.

- The programming interface is the *export interface*, which indicates what operations and data a module makes available to other modules.
- The *import interface* indicates what a module uses from other modules.

In the sequel, by interface we mean export interface unless explicitly noted otherwise.
Module as a means of abstraction

M. Reiser and N. Wirth (1992) describe the module concept as follows:

The module provides mechanisms for:

1. **structuring of a program into independent units**;
2. **declaration of variables that keep their value for the duration the module is active (that is, in memory)** – these variables are called global to the module;
3. **export of variables and procedures to be used in other modules**.

The module therefore provides facilities for abstractions.
Module as expression means for modeling

By a skillful factorization of functions, procedures and data, one can form abstractions which correspond to entities from the material world as modeled in a software system.

Examples:
- a bank account,
- the GPS navigation unit of a helicopter,
- the file system of a PC,
- the interaction elements of a graphical user interface.
Example: Module for storing error messages
DEFINITION MODULE ErrorLogger;

    FROM ErrorStream
        IMPORT ErrorMsg, ErrorType; /* Import interface */

    PROCEDURE AddErrorMsg(\em: ErrorMsg);
    PROCEDURE PrintAllErrorMsgs();
    PROCEDURE PrintErrorMsgsOfCertainType(\et: ErrorType);
    PROCEDURE ClearAll();

    . . .

END ErrorLogger.
IMPLEMENTATION MODULE ErrorLogger;

VAR errors: ARRAY [0..cMaxNoOfStoredErrors–1] OF ErrorMsg;

PROCEDURE AddErrorMsg(em: ErrorMsg)
BEGIN
  ...
  /* insert em at the next free space in the field errors */
  END AddErrorMsg;
  ...

END ErrorLogger.

ARRAY => statically established upper limit for the number of storable error messages
Advantage of the separation of interface from implementation

The implementation of the module can be improved or changed without changing the interface.

For example, in the module ErrorLogger the array structure could be replaced by a concatenated list.

The concept of *information hiding* (see the following section) contributes to a stable interface of a module.
Desired Characteristics of Modules
The design principle *Information Hiding* goes back to David L. Parnas (1972).

Thus, modules are to be designed in such a way that the data structures are hidden from the user.

Access to data and its manipulation is possible only over access procedures, which are aforementioned in the module interface.
Stable and Understandable Module Interfaces by Information Hiding (II)

- A generalization of the information hiding principle is the requirement that in the design of modules one takes care to hide as many details of the implementation as possible behind the module interface, in order not to confront the user of a module with unnecessary details and complexity.

- This can go beyond just hiding the data structures.