Università della Svizzera italiana

Faculty of Informatics

Architectural Patterns

Prof. Cesare Pautasso

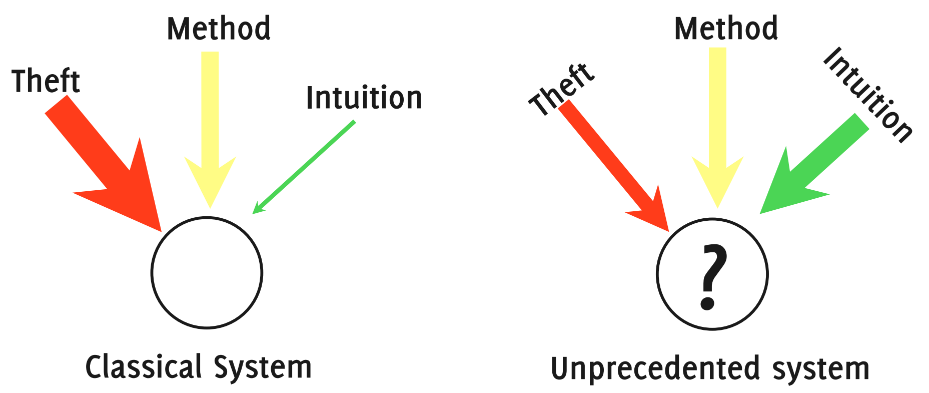
http://www.pautasso.info

cesare.pautasso@usi.ch

@pautasso

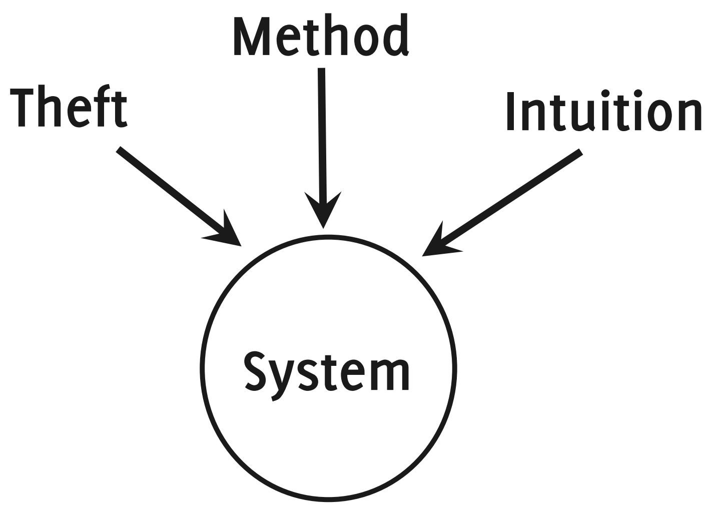
How to Design

Grady Booch



How to Design

Grady Booch



Why patterns?

Patterns help you build on the collective experience of many skilled software engineers

Defining Patterns

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

Christopher Alexander -- A Pattern Language, 1977

Patterns

Goal: practical advice and solutions to concrete design problems

A pattern is a standard solution to common and recurring design problems

Related: related (but different) to Object-Oriented Design Patterns

Hint: apply patterns where they actually solve a problem

Warning: as with all design decisions, all consequences of choosing to use a pattern should be evaluated and fully understood

Design Patterns

Capture the static and dynamic roles and relationships in solutions that occur repeatedly

Architectural Patterns

Express a fundamental structural organization for software systems that provide a set of predefined subsystems, specify their relationships, include the rules and guidelines for organizing the relationships between them

Douglas C. Schmidt

Architectural Styles

General constraints: Usually there is one dominant style for each architecture.

Architectural Patterns

Fine-grained constraints: Many patterns are usually combined. The same pattern can be used many times

George Fairbanks

Layered

1. State-Logic-Display
2. Model-View-Controller
3. Presenter-View

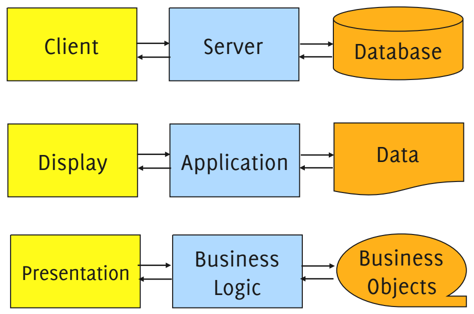
State-Logic-Display

Goal: separate elements with different rate of change

$$\boxed{\text{cluster elements that change at the same rate}}$$

Pattern: apply the layered style to separate the user interface (display/client), from the business logic (server), from the persistent state (database) of the system

State-Logic-Display



Client is any user or program that wants to perform an operation with the system

The application logic determines what the system actually does (code, rules, constraints)

The state layer deals with the organization (storage, indexing, and retrieval) of the data used by the application logic

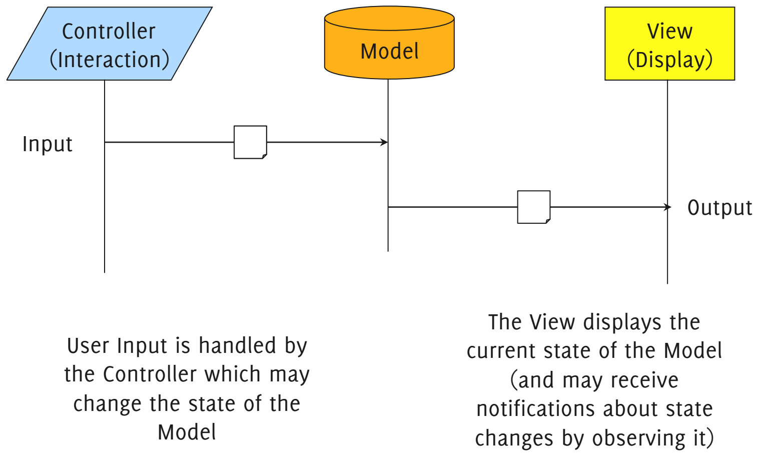
Model-View-Controller

Goal: support many interaction and display modes for the same content

separate content (model) from presentation (output) and interaction (input)

Pattern: model objects are completely ignorant of the UI. When the model changes, the views react. The controller's job is to take the user's input and figure out what to do with it. Controller and view should (mostly) not communicate directly but through the model.

Model-View-Controller


Controller (Interaction)
Model
View (Display)
Input
Output
User Input is handled by the Controller which may change the state of the Model
The View displays the current state of the Model (and may receive notifications about state changes by observing it)

Presenter-View

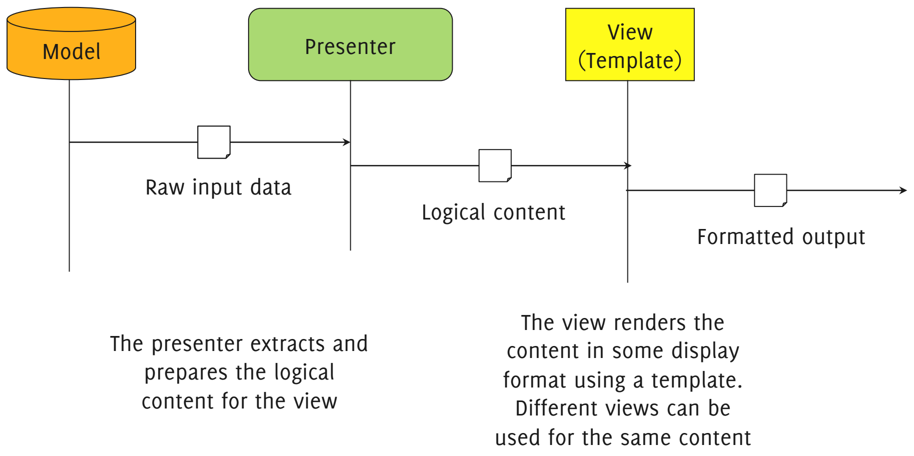
Goal: keep a consistent look and feel across a complex UI

extract the content from the model to be presented from the rendering into screens/web pages

Pattern: Apply separation of concerns. The presenter extracts and prepares the logical content to be displayed. The view formats the content consistently (possibly using templates).

Synonym: two-step View

Presenter-View


Model
Presenter
View (Template)
Raw input data
Logical content
Formatted output
The presenter extracts and prepares the logical content for the view
The view renders the content in some display format using a template. Different views can be used for the same content

Components

1. Interoperability
2. Directory
3. Dependency Injection

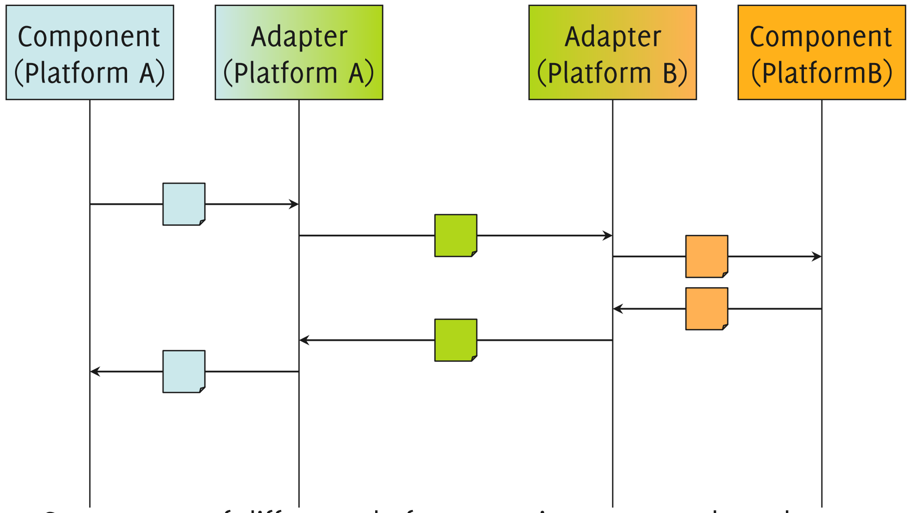
Interoperability

Goal: enable communication between different platforms

map to a standardized intermediate representation and communication style

Pattern: components of different architectural styles running on different platforms are integrated by wrapping them with adapters. These adapters enable interoperability as they implement a mapping to common means of interaction.

Interoperability



Components of different platforms can interoperate through adapters mapping the internal message format to a common representation and interaction style

Interoperability

- Why two adapters?
  - For Point to Point solutions one adapter is enough (direct mapping)
  - In general, when integrating among N different platforms, the total number of adapters is reduced by using an intermediate representation
- Adapters do not have to be re-implemented for each component, but can be generalized and shared among all components of a given platform
- This pattern helps to isolate the complexity of having to deal with a different platform when building each component. The adapters make the heterogeneity transparent.
- Warning: performance may suffer due to the overhead of applying potentially complex transformations at each adapter

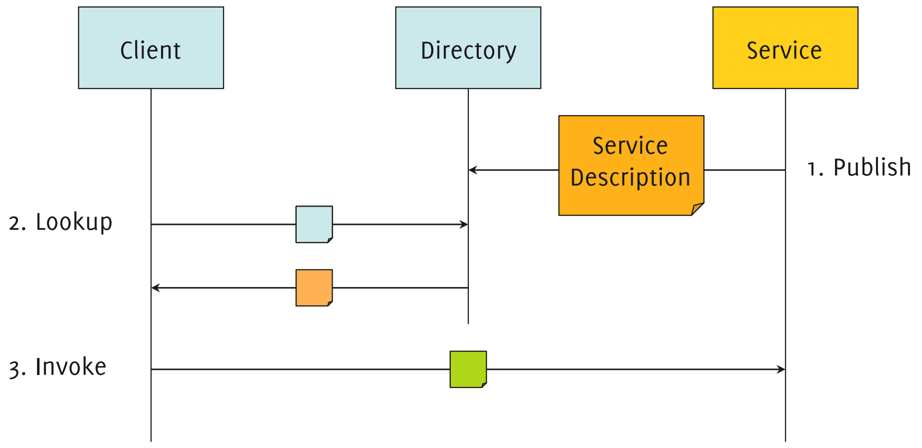
Directory

Goal: facilitate location transparency (avoid hard-coding service addresses in clients)

use a directory service to find service endpoints based on abstract descriptions

Pattern: clients lookup services through a directory in order to find out how and where to invoke them

Directory



Clients use the Directory Service to lookup published service descriptions that will enable them to perform the actual service invocation

Directory

- Clients cannot directly invoke services, they first need to lookup their endpoint for every invocation.
- Directories are a service in its own right (the difference is that its endpoint should be known in advance by all clients).
- The directory data model heavily depends on the context where the directory is applied and to the common assumptions shared by clients and services:
  - Simplest model: map `Symbolic Name` to `IP:Port`
  - Directories can store complete service descriptions or only store references to service descriptions
  - Depending on the metadata stored in the directory, clients can perform sophisticated queries over the syntax and semantics of the published services
- Directories are a critical point for the performance and the fault-tolerance of the system. Clients usually cache previous results and repeat the lookup only on failure of the invocations.
- The amount of information stored in a directory may grow quite large over time and needs to be validated and kept up to date. Published services should be approved in order for clients to trust the information provided by the directory.

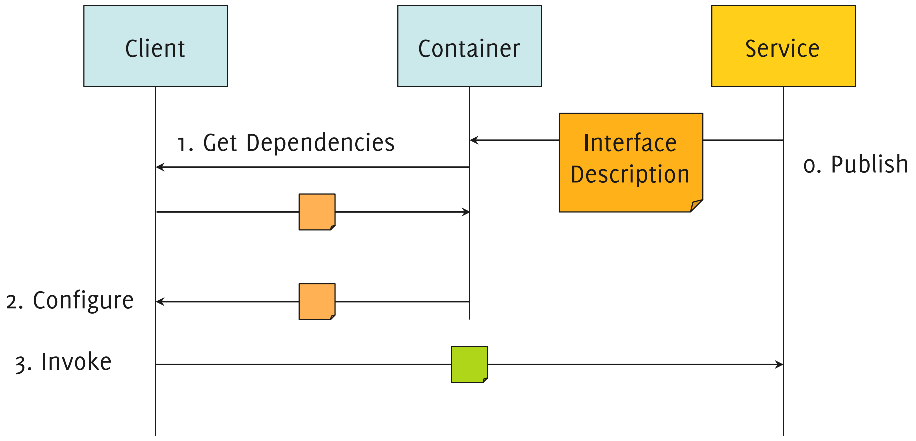
Dependency Injection

Goal: facilitate location transparency (avoid hard-coding service addresses in clients)

$$\boxed{\text{use a container which updates components with bindings to their dependencies}}$$

Pattern: clients expose a mechanism (setter or constructor) so that they can be configured

Dependency Injection



As components are deployed in the container they are updated with bindings to the services they require

Dependency Injection

- Used to design architectures that follow the inversion of control principle (“don't call us, we'll call you”, Hollywood Principle)
- Components are passively configured (as opposed to actively looking up services) to satisfy their dependencies:
  - Components should depend on required interfaces of services so that they are decoupled from the actual service implementations
- Flexibility:
  - Systems are a loosely coupled collection of components that are externally connected and configured
  - Components could be reconfigured at any time (multiple times)
- Testability: Easy to switch components with mockups