Software Architecture and Data Models

Waheedullah Sulaiman Khail[1], Pierre Schnizer[1]

[1]Helmholtz-Zentrum Berlin
waheedullah.sulaiman_khail@helmholtz-berlin.de

2nd Accelerator Middle Layer Workshop
12 – 14 February 2025 HZB/Berlin

Software architecture
Key principles
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

Table of Contents

1 What is Software Architecture?

2 Key Principles of Software Architecture

3 Common Architectural Patterns

4 What are Data Models?

5 Relationship Between Architecture and Data Models

6 Good Data Models

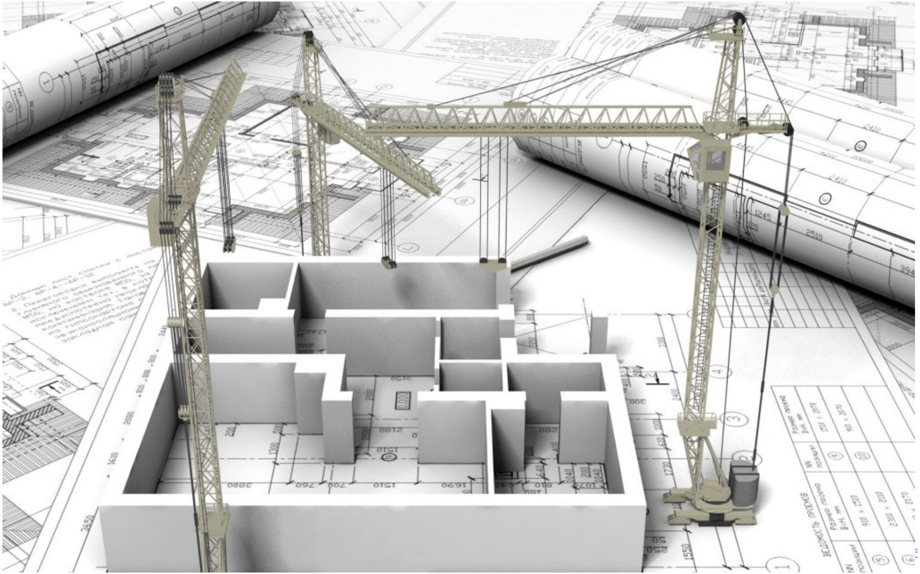
7 Common Pitfalls to Avoid

What is Software Architecture?

The high-level structure of a software system, including its components, relationships, and principles governing its design and evolution.

- Design Principles (e.g., SOLID, DRY)
- Components (e.g., modules, services)
- Relationships (e.g., APIs, dependencies)

Software architecture from and architects view



Architecture: Buildings vs. Software

- A building starts with an empty skeleton (framework), then floors, walls, and plumbing are added.
- Similarly, software starts with a high-level structure before adding modules, services, and features.
- Good architecture ensures stability, flexibility, and scalability.

Software architecture
Building Software Architecture
Why Software Architecture?
Key principles
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

Why Software Architecture Matters

- Simplifies development: Easier to understand and modify.
- Encourages modularity: Components are independent and reusable.
- Enhances scalability: Allows for adding features without breaking the system.

What is Software Architecture?

The high-level structure of a software system, including its components, relationships, and principles governing its design and evolution.

- Design Principles (e.g., SOLID, DRY)
- Components (e.g., modules, services)
- Relationships (e.g., APIs, dependencies)

Key Principles of Software Architecture

SOLID Principles:

- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

Other Principles:

- DRY (Don't Repeat Yourself)
- KISS (Keep It Simple, Stupid)
- YAGNI (You Aren't Gonna Need It)

SOLID: Single Responsibility Principle

A class should have only one reason to change:

- Easier Maintenance – Each class does only one thing, doesn't affect others.
- Better Readability – Easier to understand.
- Improved Testability – e.g. Unit testing
- Loose Coupling – Less dependent on each other.

SOLID: Single Responsibility Principle

IOC Server

initialize_pv()
update_pv()
run_server()
monitor_system()
all within one entity

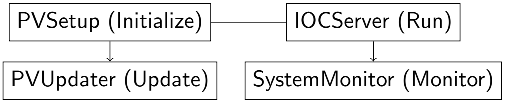
IOC Server

```
class MonolithicServer:
    def __init__(self):
        self.pvs = {}

    def initialize_pv(self,
    ↪   pv_name, value=0.0):
        self.pvs[pv_name] = value

    def update_pv(self, pv_name,
    ↪   new_value):
        if pv_name in self.pvs:
            self.pvs[pv_name] =
            ↪   new_value

    def run_server(self):
        print("Starting the
        ↪   Server...")

    def monitor_system(self):
        print("Monitoring system
        ↪   health...")
```

SOLID: Single Responsibility Principle

IOC Server

`initialize_pv()`
`update_pv()`
`run_server()`
`monitor_system()`
all within one entity



Good design

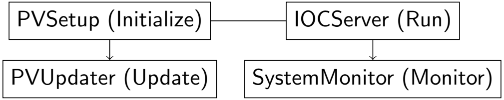
PV initialization: `PVSetup`
Updates: `PVUpdater`
Server exec: `IOCServer`
Health monitor: `SystemMonitor`

SRP: target

Separate setup / server logic
Updates independant of setup
separate server execution
Health monitoring is a separate concern

Software Architecture
W. Sulaiman Khail
Software architecture
Key principles
Single Responsibllity
Open/Closed
Liskov Substitution
Interface Segregatation
Dependency Inversion
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

SOLID: Single Responsibility Principle



Good design

PV initialization: `PVSetup`
Updates: `PVUpdater`
Server exec: `IOCServer`
Health monitor: `SystemMonitor`

SRP: target

Separate setup / server logic
Updates independant of setup
separate server execution
Health monitoring is a separate concern

SRP applied

```python
class PVSetup:
    def initialize_pv(self, pv_name):
        print(f"Initialized PV: {pv_name}")

class PVUpdater:
    def update_pv(self, pv, new_value):
        print(f"Here is update logic")

class IOCServer:
    def run(self):
        print("Server is running...")

class SystemMonitor:
    def monitor(self):
        print("Monitoring system health...")
```

SOLID: Open/Closed Principle (OCP)

Software entities (classes, functions, modules) should be open for extension but closed for modification.

- Separate pv initialization for magnets and separate update functionalities.
- Adding new magnets (types) will only need to extend pv initialization
- Handling pv update logic should be generic and by adding new magnets they shouldn't be effected.

Software architecture
Key principles
Single Responsibllity
Open/Closed
Liskov Substitution
Interface Segregatation
Dependency Inversion
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

SOLID: Liskov Substitution Principle

Rule

- *substitute* of *parent* class
- must not break behaviour of superclass

Objects of a subclass should be able to replace objects of the superclass without affecting the correctness of the program

```
class LiaisonManagerBase(metaclass=ABCMeta):
    @abstractmethod
    def forward(self, lat_elem: str, prop: str) -> (str, str):
        """Abstract translation method"""

class Pontifex(LiaisonManagerBase):
    def forward(self, lat_elem: str, prop: str) -> Sequence[(str,
    ↪   str)]:
        """Violates LSP by returning sequence of str"""
```

SOLID: Liskov Substitution Principle

```python
class ElementProxy:
    def update(self, prop, val):
        print(f"Updating {prop} with {val}")

class KickAngleCorrectorProxy(ElementProxy):
    def update(self, prop, val):
        if property_id == "x_kick":
            print(f"Applying correction for {prop}: {val}")
        else:
            # Preserves expected behavior
            super().update(prop, val)
```

Rule

- *substitute* of *parent* class
- must not break behaviour of subclass

Here: derived returns *Sequence[str,str]* instead of (str, str)(also violates SRP)

SOLID:Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use

- An interface should contain only the methods that are relevant to a specific client.
- A class should not be forced to implement methods it does not need.
- Instead of creating large, general-purpose interfaces, break them into smaller, specific interfaces.

E.g: Control system signal: split interface for *read* and *write* Have a look to ophyd-async protocol definitions

SOLID:Interface Segregation Principle (ISP)

Software architecture
Key principles
Single Responsiblity
Open/Closed
Liskov Substitution
Interface Segregatation
Dependency Inversion
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

Interface: forces bpm to be settable

```python
class SignalBase:
    def read(self):
        "reads value from control system"
    def put(self, val):
        "write value to control system"

class BPM(SignalBase):
    def read(self):
        "get beam position"
    def put(self):
        "No way to do that (yet)"
```

SOLID:Interface Segregation Principle (ISP)

Interface

```python
class Readable:
    def read(self) -> Union[float, int, Sequence[int], Sequence[float]]:
        """Reads value from control system"""
        raise NotImplementedError

class Writable:
    def put(self, val: Union[float, int, Sequence[int], Sequence[float]]) -> None:
        """Writes value synchronously to control system"""
```

SOLID:Interface Segregation Principle (ISP)

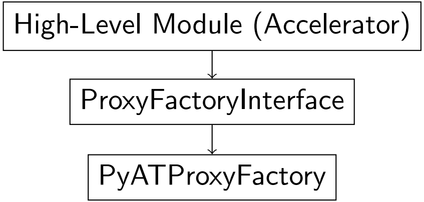
Example: BPM

```python
class Readable:
    def read(self):
        """Reads value from control system"""
        raise NotImplementedError

class Writable:
    def put(self, val) -> None:
        """Writes value synchronously to control system"""

class ReadOnlySignal(Readable):
    def read(self) -> float:
        return 42.0  #  Only needs to implement `read()`

class WriteSignal(Writable):
    def put(self, val: float) -> None:
        print(f"Sync Writing {val} to control system")  #  No async needed

class RWSignal(Readable, Writable):
    def read(self):
        return 42.0
```

SOLID: Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions

- High-level modules (business logic) should not directly depend on low-level modules (implementations like databases, APIs, etc.).
- Instead, both should depend on an abstraction (like an interface or abstract class).
- This makes code flexible, scalable, and easier to maintain.



interface

```
class TrnsltnServMgrBase(metaclass=ABCMeta):
    @abstractmethod
    def forward(self, lat_elem: str, elem_prop: str) -> TrnsltnObjBase:
        """provide corresponding device and its property"""
    @abstractmethod
    def inverse(self, dev_name: str, dev_prop: str) -> TrnsltnObjBase:
        """following Bluesky/Ophyd naming convention """
```

implementation

```
from bact_architecture import TrnsltnServMgrBase, TrnsltnObjBase
class TranslationServiceMgr(TrnltnServMgrBase):
    def __init__(self, translator: TrnsltnObjBase = Babelfish()):
        self.translator = translator
    def forward(self, elem_prop, dev_prop) -> TrnsltnObjBase:
        """provide corresponding device and its property"""
        return self.translator

    def inverse(self, elem_prop, dev_prop) -> TrnsltnObjBase:
        """provide corresponding device and its property"""
        return self.translator

class BabelFish(TrnsltnObjBase):
    """if we had one"""
```


Software architecture
Key principles
Single Responsibllity
Open/Closed
Liskov Substitution
Interface Segregatation
Dependency Inversion
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

Common Architectural Patterns

- Layered Architecture
- Monolithic Architecture
- Microservices Architecture
- Event-Driven Architecture

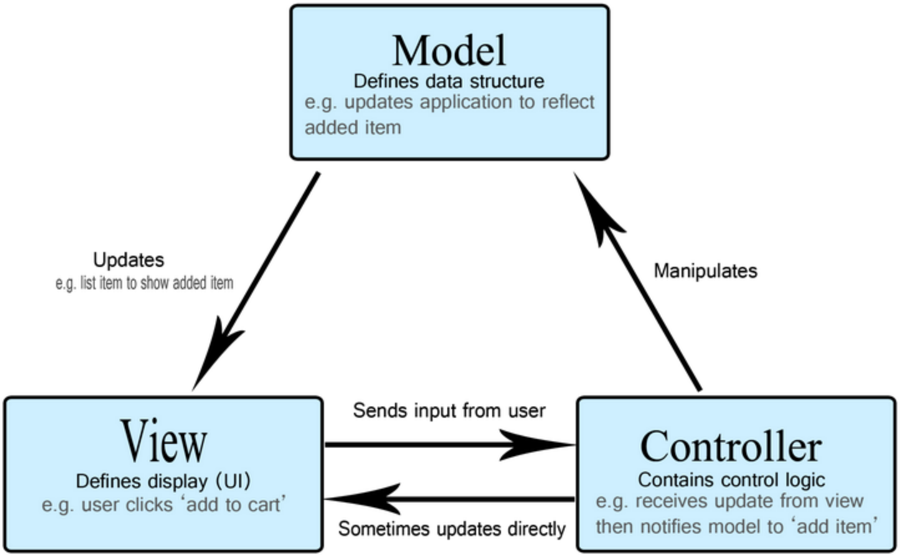
Layered Architecture

Separates concerns into different layers (UI, Business Logic, Data Access)
Best example MVC (Model-View-Controller).

- **UI Layer:** `views/`
- **Business Logic Layer:** `core/accelerators/`, `bl/`, `calculations/`
- **Data Layer:** `model/`, `ioc/`

Software architecture
Key principles
Common Architectural Patterns
Layered Architecture (N-Tier)
Monolithic Architecture
Microservices Architecture
Event-Driven Architecture
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

MVC Pattern



Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
Sends input from user
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

Monolithic Architecture

Best for: Small to medium applications, Startups
Example: Traditional Web Applications (Flask, Django)

- All components (UI, business logic, and model) are tightly coupled into a single application.
- Simple deployment, easy debugging

Microservices Architecture

Best for: Large, scalable applications (Netflix, Amazon)
Example: Cloud-based applications

- Each service handles a specific function independently.
- Communicates via APIs (HTTP, Messaging Queues).
- Scalable, independent deployments

Event-Driven Architecture

Best for: Event-driven architecture (EDA) enables decoupled, scalable, and asynchronous communication between components using events. In our accelerator system, events serve as the backbone for data flow, state updates, and process synchronization

- The accelerator model generates events when Twiss parameters, orbit data, or element values change.
- accelerator controller subscribes to events and manages their execution.
- result views listens for orbit and Twiss updates to push new values to Process Variables (PVs).
- The event model module provides a centralized mechanism for event subscriptions and broadcasting. Asyncio ensures non-blocking execution of event handlers.

Software architecture
Key principles
Common Architectural Patterns
Layered Architecture (N-Tier)
Monolithic Architecture
Microservices Architecture
Event-Driven Architecture
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

Event-Driven Architecture

Key benefits

- Loose Coupling – Components operate independently, improving modularity.
- Scalability – Additional subscribers can be added without modifying event sources.
- Asynchronous Execution – Efficient handling of high-frequency updates without blocking other operations.

What are Data Models?

A data model is an abstract representation of how data is stored, processed, and accessed in a system

- Define data structure
- Ensure data integrity
- Facilitate communication between stakeholders

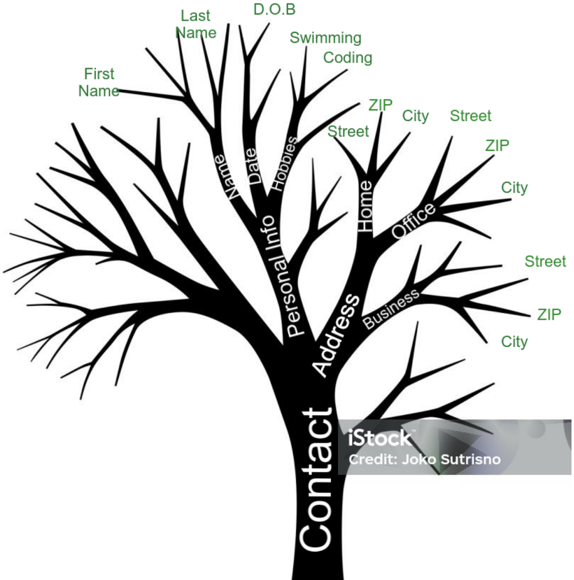
What is Software Architecture?

The high-level structure of a software system, including its components, relationships, and principles governing its design and evolution.

- Design Principles (e.g., SOLID, DRY)
- Components (e.g., modules, services)
- Relationships (e.g., APIs, dependencies)

Software architecture
Key principles
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

Contact Data Model



Software Architecture
W. Sulaiman Khail
Software architecture
Key principles
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

BBA Measurement Data in Model

BBA Measurement data model

Types of Data Models

- Conceptual Data Model: High-level, business-focused.
- Logical Data Model: Defines structure (e.g., tables, relationships).
- Physical Data Model: Implementation-specific (e.g., database schema).

Data Representation in a Data Mode

Type	Example Entities	Example Attributes
Physical Objects	Car, Book, Building	color, size, year
People & Roles	User, Admin, Customer	name, email, role
Transactions	Order, Invoice, Payment	amount, status, date
Configurations	Settings, Preferences	theme, language
Events	OrderShipped, UserLoggedIn	timestamp, details
Business Rules	DiscountPolicy, TaxCalculator	percentage, ruleset

Data Representation in a Data Model: accelerator

Type	Example Entities	Example Attributes
Physical Objects	magnets, BPM's, cavities	ID, length, K, t_f, slew rates, settle times
Events	Data arrived, ready for data	timestamp, timeout, failure, detail
Measurement Plans	setup, measurement steps, ...	device names, values,
Physical Objects	magnets, BPM's, cavities	ID, length, K, t_f
People & Roles	Operator, Shift Manager, Physist	name, email, role
Configurations	operation mode, ORM mode	settings

Relationship Between Architecture and Data Models

- Architecture defines how components interact; data models define how data is structured.
- Example: In microservices, each service may have its own data model, leading to decentralized data management.
- Challenges: Data consistency, synchronization, and scalability.

Best Practices for Software Architecture

- Design for scalability and flexibility.
- Use modular and reusable components.
- Prioritize security and performance.
- Document your architecture.

Best Practices for Data Modeling

- Normalize data to reduce redundancy.
- Denormalize for performance when necessary.
- Use indexing and partitioning for large datasets.
- Plan for data migration and versioning.

What Makes a Good Data Model?

- A good data model ensures users understand where data is and how it looks.
- Users should interact with structured information instead of raw data.
- Helps improve usability, readability, and maintainability of accelerator data.

Common Pitfalls to Avoid

- Over-engineering the architecture.
- Ignoring non-functional requirements (e.g., performance, security).
- Poor data modeling leading to inefficiencies.
- Lack of documentation.

Conclusion

Thank You!

SOLID: Single Responsibility Principle: AML bad example

Software architecture
Key principles
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

srp: sometimes

```
machine_abstraction = myMachine

class BadML:
    def put(lat_num: int,  prop: str, val: float):
        d = self.lut[lat_num]
        machine_abstraction[d[dev]] = d[prop] * val
```

SOLID: Single Responsibility Principle: AML good example

```
class GoodML(GoodMLBase):
    def __init__(self, cm: CmdRwrtrBase, mi: MachineIFBase):
        self.cm, self.mi = cm, mi
    def update(self, elem: str, prop: str, val: object, on_error: PossibleActions):
        return self.mi( self.cm(
                Command(lat_elem=elem, prop=prop, val=val, on_error=on_error)
            ) )
class CommandRewriter(CmdRwrtrBase):
    def __init__(self, lm: LiasionMgrBase, tm: TrnltnServMgrBase):
        self._lm, self._tm = lm, tm
    def forward(self, cmd: Command):
        dev_id, dev_prop = lm.forward(cmd.lat_elem, cmd.prop)
        return Command(
            dev=dev_id, prop=dev_prop,
            val=self._tm.get([cmd.lat_elem, cmd.prop], [dev_id, dev_prop]).forward(val)
        )

class LiasionMgr(LiasionMgrBase):
    def forward(self, lat_elem: str, cmd: object):
        """provide corresponding device and its property"""
```


Software architecture
Key principles
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

SOLID: Single Responsibility Principle: AML good example

Software architecture
Key principles
Common Architectural Patterns
Data models
Architecture and data models
Good Data Models
Pitfalls
Additional Slides

AML good

```
class TranslationServiceMgr(TrnltnServMgrBase):
    def forward(self, elem_prop, dev_prop) -> TranslationObject:
        """provide corresponding device and its property"""


class TranslationObject(metaclass=ABCMeta):
    @abstractmethod
    def forward(self, val: object) -> object:
        ""

class Babelfish(TranslationObject):
    """the golden grail"""
```

SOLID:Interface Segregation Principle (ISP)

```python
from bact_architecture.interfaces.signal import Readable, Writetable, AyncWritable

class EpicsROSignal(Readable):
    def read(self) -> Union[float, int, Sequence[int], Sequence[float]]:
        """implement to transport layer"""
class EpicsRWSignal(EpicsROSignal, Writable):
    def put(self, val: Union[float, int, Sequence[int], Sequence[float]]) -> None:
        """implement to transport layer"""

class PowerConverter:
    def __init__(self, setpoint: Writable, readback: Readable):
        self.setp, self.rdbk = setpoint, readback
    def read(self):
        return self.rdbk.read()
    def put(self, val):
        self.setp.put(val)
```