Topics
- UML
- class diagrams
- sequence diagrams
- communication diagrams
- design principles
- software architecture
- design patterns

Development

development: converting the system specification into an executable system

Main concern: how should the system work?

Development = Design + Implementation

development: converting the system specification into an executable system

Traditionally broken down into several stages:
- architectural design
- interface design
- abstract specification
- coding
- development is an iterative process with feedback between the stages
- design and implementation are typically interleaved

Design vs. Modeling

Design is the process of deciding how the requirements should be implemented.

- guided by design principles
- part of development

Modeling is the process of creating an abstract representation of the domain or the system.

- uses modeling languages
- spans requirements and development
UML (Unified Modeling Language)
- graphical modeling language
  - standardized by OMG (Object Management Group)
  - semi-formal
  - variety of different diagram types
- supports object-oriented designs
  - but no fixed methodology
- unified: each diagram gives a different view on the same system
- developed by Rumbaugh, Booch, Jacobson et al.
  - starting early-90’s, unification mid-90’s

UML diagram types
- Diagram
  - Structure diagram
  - Behavior diagram
  - Activity diagram
  - Use case diagram
  - State machine diagram

Structural vs. behavioral modeling
- System = structure + behavior
- Structural models show the system’s organization in terms of its components and their relationships
  - can be static (classes) or dynamic (threads)
- Behavioral models show the system’s dynamic as it is executing and responding to stimuli
  - can be events or data

Class Diagrams

Class diagram essentials
- Class diagrams describe the data found in a software system.
- Main elements:
  - Classes: represent the types of the data
  - Attributes: represent the data found in instances
  - Associations: show how instances are related
  - Generalizations: represent inheritance hierarchies
  - Operations: represent the actions by instances

Classes
- A class describes a collection of instances with common properties and similar behavior.

Employee
- Name
- Rank
- GetRank()
- SetRank()

Attributes
- Noun phrase, singular, camelCase

Operations
- Verb phrase, camelCase

More information:
A class describes a collection of instances with common properties and similar behavior.

Class diagram:
- Employee
  - name : string
  - rank : Rank
  + getRank() : Rank
  + setRank(rank : Rank) : void

Attributes and operations can have additional visibility and type information.

An association describes how instances (of two classes) reference each other.

Association diagram:
- Employee
- Company
  - workFor
  - name
  - getRank()
  - setRank()

Reading order: left entity <association> right entity.

Note: multiplicity carry a lot of meaning, so they should always be given.

Multiplicities describe how many instances (of the two classes) are related by an association.

Multiplicity diagram:
- Employee
- Company
- Board
  - workFor
  - name
  - rank
  - getRank()
  - setRank()

- “non-empty”
- “many”
- “optional”

Symbol at an association end indicates how many instances of this end are related to one instance on the other end.

Note: multiplicities are difficult to get right, so always read them in both directions.
**Typical multiplicities**
- **one-to-many**: common, strictly hierarchical relations
- **many-to-many**: common, matrix relations
- **one-to-one**: often abused

![Diagram](https://via.placeholder.com/150)

**Association directionality**
- associations are normally bi-directional:
  - for each **Order**, we know **Items** it contains
  - for each **Item**, we know which **Order** it belongs to
- association direction can be restricted (**navigable**):
  - can only go from **Order** to **Item**
  - ... but not vice versa

![Diagram](https://via.placeholder.com/150)

**Exercises**
Model with appropriate multiplicities:
1. docents teaching courses taken by students
2. musicians planning performances of different plays at different venues

**Named association ends**
- association ends can be named (**role names**)  
- associations can be reflexive

![Diagram](https://via.placeholder.com/150)

**Association classes**
- some attributes can be difficult to allocate to classes:
  - where should **grade** go?
  - use **association class**:
    - association name disappears
    - each **Student-Class** link contains an object of type **Registration**
**Association classes can be eliminated.**
- replace association + association class
  
  ![Diagram of Student, Class, Registration, Grade, int associations](image)

  *couldn’t get this working in Astah...*

  by two (one-to-many) associations:

  ![Diagram of Student, Registration, Grade, int, Class associations](image)

**Aggregation**
- Aggregations represent part-whole associations between instances.

  ![Diagram of Vehicle, Part, aggregate class](image)

  Use when
  - the parts are partOf the aggregate
  - (or the aggregate isComposedOf the parts)
  - whoever owns or controls the aggregate, also owns or control the parts

**Aggregation hierarchies**
- typically one-to-many

  ![Diagram of Vehicle, Chassis, BodyPanel, Door, Frame, Engine, Transmission, Wheel](image)

  - aggregation can be arranged hierarchically
  - NOTE: not the same as inheritance

**Composite associations**
- Composite associations represent strong part-whole associations between instances.

  ![Diagram of Building, Room, composite class](image)

  Use when
  - the part cannot exist without the whole aggregate

**Composite associations vs. attributes**
- one-to-one composite associations can also be modeled as attributes (and vice versa)

**Exercises**
- Which mechanism (association, aggregation, or composition) should be used to model the following relations:
  1. telephone and handset
  2. school and teachers
  3. book and chapters
  4. country and provinces
  5. polygon and lines
**Generalization**

Generalization represents inheritance hierarchies. This a relation on classes, not an association between instances!

- A subclass is a superclass must make sense!

Inheritance must obey the *isa rule*:
- "a subclass is a superclass" must make sense!

**Generalization**

- use a discriminator label to denote the criterion used to distinguish different subclasses
  - in particular for disjoint generalization

**Pitfalls – multiple generalization**

- UML allows multiple inheritance
  - but Java doesn’t...
  - and there are problems with repeated attributes

**Pitfalls – overgeneralization**

- no difference other than discriminator label
- better solution:

**Pitfalls – class change**

- instances should never change their class!
**Interfaces**

An **interface** describes a *portion of the visible behaviour* of a set of objects.

- similar to classes, except they lack instance variables and implemented methods

---

**Domain Class Model**

**A recipe to cook domain class models...**

1. add classes (*without* attributes)
   - identify relevant real system objects and represent as classes
2. add generalizations
3. add associations (*with* multiplicities)
   - identify relations between identified objects
4. add aggregations and compositions
   - check whether really necessary
5. add attributes (*no* operations)
   - identify relevant core attributes and add to corresponding classes
6. stir until done

---

**Class diagrams can be used to model the problem at different levels of abstraction.**

- **domain model**
  - developed in domain analysis to understand the domain
  - models also aspects of the domain that will not be implemented by the system
  - also called **exploratory domain model**
  - **shrink & refine**

- **system domain model**
  - models only aspects of the domain that are implemented by the system
  - **grow & refine**

- **system model**
  - models also classes outside the domain but used to build the user interface and system architecture

---

**Domain class model**

The **domain class model** contains:

- **relevant entities as classes:**
  - physical objects
  - persons, organizations (**actors**)  
  - events, processes, abstractions
- **links between entities as associations:**
  - relations
  - communications
  - part/whole relations (**aggregation, composition**)
**Discovering domain classes**

**Noun phrase analysis:**
- analyze (textual) documents describing the system
  - use cases
  - requirements documents
- extract the nouns and noun phrases
- eliminate nouns that
  - are redundant, vague, or highly general
  - are too specific or represent specific instances
  - refer to the entire system or objects outside the application
- pay attention to nouns that describe different user types or other actors

**Discovering associations**
- start with central and most important classes
- work outwards towards the less important classes
- add an association if one class
  - possesses or controls
  - is related to
  - communicates with
  - is a part of
  - is a member of
  - some other class in the model
- label it clearly
- specify the multiplicity at both ends
- KISS: keep it simple

**Pitfalls – transient associations**
- an association is only legitimate if the links "survive" beyond execution of an operation
  - links are stored in database
  - if nothing needs to be stored, rethink the association

**Pitfalls – actions as associations**
- actions should not be modelled as associations but as association classes
  - store information associated with action
  - different operations access different attributes
Pitfalls – actions as associations

- **actions** should **not** be modelled as associations but as **association classes**
  - store information associated with action
  - different operations access different attributes

Pitfalls – wrong multiplicities

Sanity-check the multiplicities with a few questions:

- Do we model **generic / indistinguishable** or **individual / distinguishable** items?

Pitfalls – wrong multiplicities

Sanity-check the multiplicities with a few questions:

- Do we model **generic / indistinguishable** or **individual / distinguishable** items?
- Do we model a **static view (snapshot)** or a **dynamic view (history)**?
- Do we model **initial** or **exceptional** situations correctly?

Discovering attributes

- information that must be maintained in each class
  - nouns rejected as classes may become attributes
- attributes should generally contain a simple value
  - string, number, date, ...
- if a subset of a class’s attributes form a coherent group, then create a new class from these attributes
Example: meal ordering system

Pitfalls – repeated attributes

It is not good to have repeated attributes:

Discovering generalizations / interfaces

There are two ways to identify generalizations:

- **bottom-up**: group together similar classes creating a new superclass
- **top-down**: look for more general classes first, specialize them if needed

Create an interface, instead of a superclass if:

- some of the classes already have superclasses
- the classes are very dissimilar except for having a few operations in common
- different implementations of the same class might be available

Data Dictionary

Goals of the data dictionary

- **find** terms quickly
  - sort and cross-reference entries
- **understand** concepts without knowledge of UML
  - use natural language descriptions
  - ensure consistency with diagrams
- **eliminate** ambiguities
  - give precise definitions
  - use terms consistently
  - list commonly used alternative terms
- **explain** design decisions

Data dictionary

The **data dictionary** is a **centralized repository** of information about data such as meaning, relations to other data, origin, usage, and format.

- **glossary** that describes artefacts from diagrams
- contains **descriptions** and **restrictions** that cannot be expressed (easily) within the diagrams
- serves as **communication basis** between the different **stakeholders**

Source: Lethbridge/Laganiere, Object-Oriented Software Engineering
Elements of the data dictionary

- **use case diagrams**: description of **actors**
  - including different scenarios and exceptions

- **domain class model**: description of **classes**
  - including properties and attributes
  - description of **associations**
    - discussion of multiplicities

---

Example: meal ordering system

**Menu (class)**:
A list that contains the **dishes** for a day. The menu is created by the **kitchen staff**, and released by the **supervisor**. The system only contains one menu at a time.
Possible attributes: date

**orders (association)** between **Patient** and **Dish**
Describes which types of the different **dishes** a **patient** has ordered for the following day. Each **patient** can normally order one **breakfast**, one **lunch**, and one **dinner**; if they get released the following day, they can only order **breakfast**. Each **dish** can be ordered by many **patients**. [...]

**Supervisor (actor, class)**
[...]

---

Recap – what have we modeled so far?

- **use case diagrams** to capture the interactions between actors and system
  - dynamic, high-level system model

- **class diagrams** to capture the domain structure
  - static, high-level domain model

Neither of the two diagram types reflects:

- the **temporal order** of interactions
- the **internal communication** structure

---

Sequence Diagrams

**Sequence diagram essentials**

Sequence diagrams describe the interactions of related **objects** in **temporal order**.

Main elements:
- **lifeline boxes**: represent the interacting **objects**
- **lifelines**: represent the **temporal order**
- **messages**: represent the object **interactions**
- **control boxes**: represent the object **interactions**

Modeling actor-system interactions

Sequence diagrams can be used to model the scenarios contained in the use cases:

- **communication** between actor and system
- **system operations** (i.e., user requests)
- **system events** (i.e., system responses)
- sequential **order** of operations and events
Modeling actor-system interactions

Sequence diagram notation

- **lifeline box** (actor)
- **lifeline box** (object)
- **operation** (message)
- **event** (message)
- **system operation**
- **system event**
- **actors**
- **system**
- **messages** can have parameters

**Lifeline boxes** represent the interacting **objects**:
- **named** object of given class (default)
- **anonymous** object of given class
- **object without class**
- **actor**

Sequence diagram notation

- **synchronous** communication
- **asynchronous** communication

**Activity zones** can express nested control flows

- **C must respond** before **B**

Sequence diagram notation

- **alternative**
- **option**

- one of three possible responses happens (depending on conditions)
- one to three possible responses happen (depending on conditions)

Sequence diagrams – further notation

- **loops**
- **object creation, object deletion**
- **not required to model actor-system interaction**
A recipe to cook sequence diagrams...

1. identify scenarios
   - look at use cases
   - rule of thumb: multiple scenarios from each use case
2. for each scenario:
   1. identify the actors
   2. name the (single) system operation
      - system operations should be independent
      - use activity diagrams to compose
   3. identify the parameters
   4. model the sequence of system events
      - find expressive names for the events
   5. add exceptions / error cases
      - use alt / opt

Example: meal ordering system

3. [Create menu] The system shall allow the kitchen staff to create menu for the following day:
   1. System shall allow kitchen staff to add/delete meals to not released menu
   2. System shall allow kitchen staff to release menu
   3. System shall ensure when releasing the menu that:
      1. menu contains at least one dish for each meal
      2. menu has a maximum of 4 dishes for any meal
   4. System shall ensure release of menu for the following day after the current day ordering process has been finished and before 8am the following day
      1. System shall ensure that released menus cannot be changed
      2. System shall provide same menu to all wards after release
      3. System shall inform ward supervisors about new menu release

Activity diagram essentials

Activity diagrams describe the stepwise flow of activities and actions.

Main purpose:
- model both organisational (i.e., workflows) and computational processes
- identify candidate system use cases
  - through the examination of business workflows
- model flow between (or within) use cases
  - combine all sequence diagrams for a single actor/use case
- identify pre- and post-conditions of use cases

Activity Diagrams
Activity diagram essentials

Activity diagrams describe the stepwise flow of activities and actions.

Main elements:
- actions / activities: represent the executable steps
- transitions: represent the control flow
- split / merge: represent decisions
- fork / join: represent concurrency
- swim lanes: structure the control flow

Activity diagram – basic structure

Origin:
- flow charts
- program execution plans

Activities:
- sequence of actions
- transitions immediate (triggerless)
- exit at final state

Activity diagram – actions vs. activities

actions:
- executable atomic computation in the control flow
- cannot be decomposed

activities:
- also executable units in the control flow but non-atomic
- can be decomposed
- can be represented by a separate activity diagram

Activity diagram – decisions

decision node:
- only one branch can execute
- use mutually exclusive conditions – [else] possible

merge node:
- reunite alternative control flows

Activity diagram – concurrency

split (fork):
- B and C only start after A
- B and C mutually independent
- both branches execute

synchronization (join):
- D happens only after B and C
- independent control flows are synchronized

Activity diagram – termination nodes

activity final:
- entire activity is ended
- including other flows

flow final:
- only current flow is ended
- other flows can continue
**Activity diagram – swim lanes**

Use swim lanes to structure activity diagrams:
- one lane per class
- split/merge and fork/join can span multiple lanes

**A recipe to cook activity diagrams...**

1. **create one activity diagram for each** actor / use case combination
   - merge all scenarios (i.e., sequence diagrams) that are triggered by each actor
2. **refine** the system operations
   - add internal activities and transitions
   - add GUI activities
   - important: restrict to actor’s point of view
3. **integrate** system events
4. **integrate** conditions / nesting
   - take guards from sequence diagrams

---

**Example: meal ordering system**

6. **[Take orders]** System shall let nurse take orders of patients for the following day:
   1. System shall only allow taking orders from released menu
   2. System shall only allow nurses to take order if patient is from their ward
   3. System shall check the patients availability
      1. System shall ensure sure that patients not discharged the following day order exactly three meals
      2. One breakfast, one lunch, one dinner
   4. System shall save the ID of nurse taking the order
   5. System shall ensure that orders cannot be changed after sending them to the kitchen
   6. System shall ensure that orders are never deleted

**Example: meal ordering system**

6. **[Take orders] System shall ensure that patients not discharged the following day order exactly three meals**
   1. One breakfast, one lunch, one dinner
   2. System shall make sure that patients that are discharged the following day can only order breakfast

---

**Example: meal ordering system**

6.3 System shall check the patients availability
   1. System shall ensure sure that patients not discharged the following day order exactly three meals
   2. One breakfast, one lunch, one dinner
   3. System shall make sure that patients that are discharged the following day can only order breakfast

**Example: meal ordering system**

6.3 System shall check the patients availability
   1. One breakfast, one lunch, one dinner
   2. System shall make sure that patients that are discharged the following day can only order breakfast
Example: meal ordering system

System Class Models

Class diagrams can be used to model the problem at different levels of abstraction.
- **Domain model**
  - developed in domain analysis to understand the domain
  - models also aspects of the domain that will not be implemented by the system
  - also called **exploratory domain model**
- **System domain model**
  - models only aspects of the domain that are implemented by the system
- **System model**
  - models also classes outside the domain but used to build the user interface and system architecture

System class model development
- determine the **system boundary**
  - actors are outside the system
- determine the **system tiers**
  - **presentation tier**
  - **application tier** (middle tier, business logic)
  - **data tier**
- use UML stereotypes to denote which tier a class belongs to

**Derive the system class model systematically from domain class model.**

<<Boundary>> classes

Boundary classes constitute the interface between system and environment:
- **Presentation layer** (GUI)
- interfaces to other systems
- sensors and switches to control external devices
  - actors communicate with the system only via boundary classes

**Design rule: one boundary class per actor.**
Control classes orchestrate the system operations:
- application layer
- encapsulate business processes and business logic
⇒ “glue” between boundary and entity classes

Entity classes manage the application data and the internal system state:
- data layer (persistence, database system)
- includes access methods
- data for actors must be reflected into the system via additional entity classes
⇒ connected to control and other entity classes via relations

Design rule: one control class per use case.

Layered architecture

Associations in the system class model

A recipe to cook system class models...

1. start with the domain class model
2. identify actors
   – check in the use case diagrams
3. identify boundary classes for actors
   – represent the user interface
4. identify entity classes
   – map properties into attributes
5. insert entity classes for actors (if required)
   – reflect necessary properties of the actors in the system...
A recipe to cook system class models...

6. identify control classes for use cases
   – between boundary and entity classes
   – typically one control class per use case
7. ensure 1:1 associations between actor/boundary and boundary/control classes
   – ensure that actors only talk to boundary classes
8. check model for completeness
   – insert new associations (if necessary)
   – model might differ structurally from domain class model
9. complete attributes in all classes

Example: boundary classes for actors

Example: actors

Insert a new boundary class for each actor, or repurpose an existing class!

Example: actors?

Check the use case diagrams for actors!

Example: boundary classes for actors?
A recipe to cook system class models...

1. start with the domain class model
2. identify actors
   - check the use case diagrams
3. identify boundary classes for actors
   - represent the user interface
4. identify entity classes
   - map properties into attributes
5. insert entity classes for actors (if required)
6. identify control classes for use cases
   - between boundary and entity classes
   - typically one control class per use case
7. ensure 1:1 associations between actor/boundary and boundary/control classes
8. check model for completeness
   - insert new associations (if necessary)
9. complete attributes in all classes

Example: entity classes

Example: entity classes for actors

Insert entity classes for actors if the system needs to keep actor data!

Classes that are not actor or boundary classes are entity classes!
Example: control classes for use cases?

Insert a control class for each use case and connect with the boundary classes according to the use case / actor relation!

Example: control classes for use cases

Insert a control class for each use case and connect with the boundary classes according to the use case / actor relation!

Example: enforce 1:1 relations

Move associations from <<actor>> to corresponding <<control>>!

irrelevant for requirements

Example: enforce 1:1 relations

Move associations from <<actor>> to corresponding <<control>>!

all of the supervisor’s associations are in control – replace by actor!

Example: enforce 1:1 relations

Move associations from <<actor>> to corresponding <<control>>!

same procedure for kitchen staff

Example: enforce 1:1 relations

Move associations from <<actor>> to corresponding <<control>>!

integrate NurseData into system; change direction and multiplicity of association

Example: enforce 1:1 relations

integrate NurseData into system; move worksOn association from <<actor>> to <<entity>>
A recipe to cook system class models...

1. start with the domain class model
2. identify actors
   - check in the use case diagrams
3. identify boundary classes for actors
   - represent the user interface
4. identify entity classes
   - map properties into attributes
5. insert entity classes for actors (if required)
   - reflect necessary properties of the actors in the system
6. identify control classes for use cases
   - between boundary and entity classes
   - typically one control class per use case
7. ensure 1:1 associations between actor/boundary and boundary/control classes
8. check model for completeness
   - insert new associations (if necessary)
   - model might differ structurally from domain class model
9. complete attributes in all classes

Example: check diagram

Example: complete attributes

Result: system class model

Recap – what have we modeled so far?

- use case diagrams and sequence diagrams to capture the interactions between actors and system
- class diagrams to capture the domain and system structure
- activity diagrams to capture the step-wise flow of activities and actions

None of the diagrams focuses on the level of individual objects.
Communication diagram essentials

Communication diagrams describe the flow of communications between objects along the associations.

Main purpose:
- more detailed model of system operations
  - serves as blue-print for implementation

Design rule: one communication diagram per system operation.

Message format

number: [guard] variable := name(parameters)

- messages have name and parameters
- message send can be guarded
  - guard must be checked locally
  - message only sent if guard evaluates to true
- messages can store results in variables
  - variables local (typically in control object)
- messages ordered by number

A recipe to cook communication diagrams...

1. create one communication diagram for each system operation
2. identify actor and control object
   - identify actor in sequence diagram
   - identify control object in system class model
   - ignore communication between boundary and control
3. identify the system operation
   - identify name and parameters
   - ensure consistency with sequence diagram
...

A recipe to cook communication diagrams...

4. identify collaborators
   - follow association links in system class model
   - introduce collections for set-valued relations
5. derive message flow
   - define messages to
     ▶ process and store data in objects
     ▶ create and delete objects
     ▶ response messages (cf. sequence diagram)
   - define order of messages (incl. conditionals)
     ▶ take pre-conditions into account
6. update system class model if necessary
   - add links and attributes

Example: change seats on booking

Change number of seats on booked public tour
- An existing booking can be changed if the scheduler enters the booking number and the new number of seats.
- If there are not enough seats available, the scheduler is notified, and nothing changes;
- otherwise, the booking is changed and scheduler is notified.

from the requirements
Example: change seats on booking

2. **identify** actor and control object
   - **identify** actor in sequence diagram
   - **identify** control object in system class model

Example: change seats on booking

3. **identify** the system operation
   - **identify** name and parameters
   - ensure consistency with sequence diagram

Example: change seats on booking

4. **identify** collaborators
   - follow association links in system class model
   - introduce collections for set-valued relations

Example: change seats on booking

Second step:

From the Booking we can access the Tour.

The BookingSystem needs to maintain several added bookings in a collection. It uses the booking number to find the corresponding booking (attribute nr) in the collection.
Example: change seats on booking

Third step:

From the Tour we can access the Bus (and its capacity and current load).

Example: change seats on booking

Fourth step:

4. derive message flow
   - define messages to
     ▶ process and store data in objects
     ▶ create and delete objects
     ▶ response messages (cf. sequence diagram)
   - define order of messages (incl. conditionals)
   ▶ take pre-conditions into account

Example: change seats on booking

Fifth step:

Final step – update booking and acknowledge:
Example: change seats on booking

6. **update** system class model if necessary
   - add links and attributes

Message flow

- message flow starts with call of system operation
- receiver is control object
- messages are **only** sent as reaction to receiving another message
- messages can **only** be sent along links that are instances of associations in the system class model
  - update class model if missing
- variables are always local
  - typically in the control object

Implementation Class Models

Class diagrams can be used to model the problem at different levels of abstraction.

- **domain model**
  - models also aspects of the domain that will not be implemented by the system
- **system domain model**
  - models only aspects of the domain that are implemented by the system
- **system model**
  - models also classes outside the domain but used to build the user interface and system architecture
- **implementation model**
  - represents system model using programming language constructs

Recap – what have we modeled so far?

- **use case diagrams** and **sequence diagrams** to capture the interactions between actors and system
- **class diagrams** to capture the domain and system structure
- **activity diagrams** to capture the step-wise flow of activities and actions
- **communication diagrams** to capture the flow of communications between objects along the associations

None of the diagrams is aimed at the implementation level.

Implementation class models serve as foundation for the implementation.

Goal: **systematic derivation** of the implementation classes from system class model.

- contains **complete description** of all class elements (attributes and methods)
  - types, visibility, multiplicities
- still uses UML syntax...
- ...but relies on constructs available in the programming language
A recipe to cook implementation class models...

1. **identify** attributes
   - retain all attributes from (updated) system class model
   - represent all associations as attributes
     - use navigation order to identify host class

2. **identify** methods
   - represent messages from communication diagrams as methods
     - use navigation order to identify host class
     - control classes host system operations

3. **determine types and properties** of attributes and methods
   - including visibility

4. **refactor** class diagram to match programming language
   - UML != Java...
   - replace association attributes by references
   - replace multiple inheritance by interfaces

---

Example: Bus booking system

1. **identify** attributes
   - retain all attributes from (updated) system class model
   - represent all associations as attributes
     - use navigation order to identify host class

3. **determine types and properties** of attributes and methods
   - including visibility

---

Extended syntax for attributes

![Extended syntax for attributes](image_url)

---

Unidirectional associations

Communication happens only in one direction:

```
A: A
B: B
```

A is client, B is server
(or: A "knows" B, B "doesn’t know" A)
Realizing unidirectional associations

Optional references:

```java
public class A {
  private B b;
  public void meth(B b) {
    this.b = b;
  }
  ...
}

public class B {
  ...
  public String print() {
    return "hello B";
  }
  ...
}
```

Implementation in Java

```
implementation in Java
```

Realizing unidirectional associations

Mandatory references:

```java
public class A {
  private B b;
  public void meth(B b) {
    this.b = b;
  }
  ...
}

public class B {
  ...
  public String print() {
    return "hello B";
  }
  ...
}
```

Implementation in Java

```
implementation in Java
```

Realizing unidirectional associations

Multiple references (fixed number):

```java
public class A {
  private Collection<B> bs;
  public void add(B b) { bs.add(b); }
  public void remove(B b) { bs.remove(b); }
}
```

Implementation in Java

```
import java.util.ArrayList;
import java.util.Collection;

public class A {
  private Collection<B> bs = new ArrayList<>();
  public void add(B b) { bs.add(b); }
  public void remove(B b) { bs.remove(b); }
}
```

Realizing unidirectional associations

Multiple references (variable number):

```java
public class A {
  private Collection<B> bs;
  public void add(B b) { bs.add(b); }
  public void remove(B b) { bs.remove(b); }
}
```

Implementation in Java

```
import java.util.ArrayList;
import java.util.Collection;
```

Realizing unidirectional associations

Multiple references (variable number):

```java
public class A {
  private Collection<B> bs;
  public void add(B b) { bs.add(b); }
  public void remove(B b) { bs.remove(b); }
}
```

Bidirectional associations

Communication happens in both directions:

```
 bidirectional
```

```
association is undirected
```

```
messages are sent in both directions
```

```
messages are sent in both directions
```

```
messages are sent in both directions
```

```
messages are sent in both directions
```

```
messages are sent in both directions
```

```
messages are sent in both directions
```

```
messages are sent in both directions
```

```
messages are sent in both directions
```

```
messages are sent in both directions
```

```
messages are sent in both directions
```
Generating class skeletons

```java
public class Buchung {
    private int sitzplaetze;
    private int nr;
    private Tour tour;

    public int getNr() {
        return nr;
    }

    public Tour getTour() {
        return tour;
    }

    public int getSitzplaetze() {
        return sitzplaetze;
    }

    public void setSitzplaetze(int plaezte) {
        this.sitzplaetze = plaezte;
    }
}
```

generator/ setter can be auto-generated

Example: Bus booking

Methods can be read of the communication diagrams as methods

1. represent messages from communication diagrams as methods
2. navigate order to identify host class
3. host system operations in control classes

```
public int getBusPlaetze() {
    return nr;
}
```

```
public class Buchung {
    private Collection<Buchung> buchungen = ...;

    public Tour getTour() {
        return tour;
    }
}
```

public class Reisetour extends Tour {
    private int sitzplaetze;

    public int getSitzplaetze() {
        return sitzplaetze;
    }
}

Implement generalization through inheritance

```
public class Tour {
    private int belegtePlaetze;
    private int nr;

    public int getBelegtePlaetze() {
        return belegtePlaetze;
    }

    public void platzwekndern (int altplaetze, int neuePlaetze) {
    }
}
```

Deriving Code from Models

Realizing bidirectional associations

system class model

implementation class model

Implementation in Java more complicated – constraints...
Generating functions

public void gebuchtePlaetzeEinerOeffentlichenReisebustourAndern(int bNr, int neuePlaetze){
    Tour t = b.getTour();
    int belegtePlaetze = t.getBelegtePlaetze();
    int alterPlaetze = t.getBusPlaetze();
    int busPlaetze = t.getBusPlaetze();
    if (! (belegtePlaetze + neuePlaetze - alterPlaetze <= busPlaetze))
        calls from diagram
    altePlaetze = b.getSitzplaetze();
    ...
Generating functions

```java
public Buchung b = findeBuchung(bNr);
if (! (belegtePlaetze + neuePlaetze - altePlaetze <= busPlaetze))
    return Systemereignis.AENDERUNG_AUS_PLATZGRUENDEN_NICHT_MOEGLICH;
int altePlaetze = b.getSitzplaetze();
```

Guard holds implicitly (because of return)

```java
public Systemereignis
gedachtePlaetzeInEineOffentlicheReisebustourAendern(int bNr, int neuePlaetze)
    Buchung b = findeBuchung(bNr);
    Tour t = b.getTour();
    int belegtePlaetze = t.getBelegtePlaetze();
    int busPlaetze = t.getBusPlaetze();
    int alterPlaetze = b.getPlaetze();
    if (! (belegtePlaetze + neuePlaetze - alterPlaetze <= busPlaetze))
        return Systemereignis.AENDERUNG_AUS_PLATZGRUENDEN_NICHT_MOEGLICH;
    b.setPlaetze(neuePlaetze);
    return Systemereignis.AENDERUNG_AUS_PLATZGRUENDEN_NICHT_MOEGLICH;
```

Guard holds implicitly (because of return)
public class Tour {
    private int belegtePlaetze;
    public void plaeetzeAendern(int altePlaetze, int neuePlaetze) {
        belegtePlaetze = belegtePlaetze + neuePlaetze - altePlaetze;
        …
        return Systemereignis;
        if (! (belegtePlaetze + neuePlaetze - altePlaetze <= busPlaetze))
            return Systemereignis.
    }
}

Buchung b = findeBuchung(bNr);
…

t.setSitzplaetze(neuePlaetze);

t.plaeetzeAendern(altePlaetze, neuePlaetze);

Software Design Principles

prin-ci-ple (noun):
1. a basic truth or theory;
2. an idea that forms the basis of something.

Software design principles...
• are abstract guidelines
• become practice through methods and techniques
  – often methods and techniques are packaged in a methodology
  – methodologies can be enforced by tools
• apply to process and product
Key design principles

- Rigor and formality
- Separation of concerns
- Modularity
  - coupling and cohesion
- Abstraction
  - information hiding
  - hierarchical structure
- Design for change
- Generality
- Incrementality

Key principle #1: Rigor and formality

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software

- even creative activities (e.g., design, programming) must be practiced systematically
- rigor: the quality of being very exact, careful, or strict
  - any systematic approach
- formality: rigor at the highest degree
  - typically mathematical methods

Key principle #1: Rigor and formality

Examples (product):
- systematic (rigorous) transformation of models
- systematic (rigorous) construction of test cases
- mathematical (formal) program correctness proofs

Examples (process):
- documentation of development steps
- cleanroom software development

Key principle #2: Separation of concerns

- aka "divide and conquer"
- develop software in a way that the issues can be addressed one at a time
- supports parallelization of efforts and separation of responsibilities

Key principle #2: Separation of concerns

Examples (product):
- different phases of a compiler
- protocol stack
- model-view-controller architecture

Examples (process):
- waterfall model
- lazy code optimization

Key principle #3: Modularity

- modularity is separation of functional concerns
- modules
  - enforce logical boundaries
  - separate interface from implementation
- modular languages support separate compilation and deployment (modules as components)
- modularity is the cornerstone of software design:

"Modularity is the single attribute of software that allows a program to be intellectually manageable."

G. J. Myers
Key principle #3: Modularity

Informally, modularity means decomposition into subprograms and tasks.

Problem: this is difficult to achieve:
- What should be a subprogram or task and why?
- What items should be parameters?
- What items should be “global” variables?
- What should be in the “main” program?

Cohesion and coupling

- modules should be highly cohesive
  - items in a module are closely related to one another
  - modules understandable as a meaningful unit
- modules should exhibit low coupling
  - modules have low interactions with others
  - modules understandable separately

Bad:

Cohesion levels

- coincidental cohesion (low)
  - module items grouped randomly
- logical cohesion (low)
  - module items perform similar functions
- temporal cohesion (low)
  - module items are activated at the same time
- communicational cohesion (medium)
  - all module items operate on the same input or produce the same output
- sequential cohesion (medium)
  - one module item’s output is another one’s input

Cohesion and coupling

- high cohesion and low coupling ⇒ simple interfaces
  - simpler communication
  - simpler correctness proofs
  - changes influence other modules less often
  - reusability increases
  - comprehensibility improves

Cohesion levels

- functional cohesion (high)
  - each item is necessary for the execution of a single function
- object cohesion (high)
  - each operation provides functionality which allows object attributes to be modified or inspected
  - inheritance weakens cohesion
  - to understand a component, the super-classes as well as the component class must be examined

Cohesion is not formally defined and often difficult to classify.

Coupling mechanisms

- content or pathological coupling (high)
  - one module modifies or relies on the internals of another module (e.g., accessing local data)
  - changing the way the second module produces data requires changing the dependent module
- common or global coupling (high)
  - two modules share the same global data (e.g., a global variable)
  - changing the shared resource requires changing all the modules using it
- external coupling (high)
  - two modules share an externally imposed data format, communication protocol, or device interface
### Coupling mechanisms

- **control coupling** (medium)
  - one module controls the flow of another (e.g., passing a what-to-do flag)
- **data coupling** (medium)
  - modules share data only through parameters
- **message coupling** (low)
  - achieved by state decentralization (as in objects) and communication via parameters or message passing

**High coupling leads to ripple effects when the code is changed.**

### Key principle #4: Abstraction

- **abstraction is separation of hierarchical concerns**
- **abstraction ignores details**
  - type and scale of abstraction depends on purpose
- **abstraction produces models**
  - (see UML)
  - trades reasoning about the system by reasoning about the model
- **procedural abstraction**: stepwise refinement
- **data abstraction**: find hierarchy in the data
  - application-oriented data structures
  - general data structures

### Information hiding

**Combination of abstraction and modularity**

*All information about a module (and particularly how the module works) should be private to the module, unless it is specifically declared otherwise.*

- world can only see module through interface
- anything beyond that interface should be hidden
  - justification: restrict changes to one location.
  - real purpose: hide design decisions

### Key principle #5: Design for change

- change of algorithms
  - e.g., replace inefficient sorting algorithm
- change of data representation
  - e.g., from custom format to XML
  - 17% of maintenance costs attributed to data representation changes
- change of underlying hardware or abstract machine
  - new devices
  - new release of operating system or DBMS
- change of “social” environment
  - new tax regime
  - EURO vs national currency in EU

**Change is inevitable, better plan for it!**

⇒ use abstraction and modularity!
**Key principle #6: Generality**
- see whether the current problem is an instance of a more general problem whose solution can be reused in other cases
- carefully balance generality against performance and cost
- sometimes the general problem is easier to solve than a special case

**Key principle #7: Incrementality**
Development should proceed in a stepwise fashion (increments):
- deliver subsets of a system early to get early feedback from expected users, then add new features incrementally
  - design for change
- deal first with functionality, then turn to performance
  - separation of concerns
- deliver a first prototype and then incrementally add effort to turn prototype into product

**Software Architecture**

**Software architecture by example...**

**Software architecture as abstraction**
Software architecture as abstraction

Software architecture – Definition #1

"The architecture of a software system defines that system in terms of computational components and interactions among those components."

M. Shaw and D. Garlan

statement

procedure

module

(design) pattern

architecture

Problem: purely static view

More information:

Software architecture – Definition #2

"The software architecture of a system is the [...] structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

Bass, Clements, and Kazman

Difference to previous definition:
- multiple system structures
- externally visible properties of components

More information:

Architectural structures

- module structure
- conceptual, or logical structure
- process, or coordination structure
- physical structure
- uses structure
- calls structure
- data flow
- control flow
- class structure

Architecture is a set of abstractions.

Other views

- Architecture is conceptual, high-level design
- Architecture is overall structure of the system
- Architecture is the structure, including the principles and guidelines governing their design and evolution over time
- Architecture is components and connectors
- Architecture is the process of designing the global organization of a software system, including:
  - dividing software into subsystems
  - deciding how these will interact
  - determining their interfaces
Why is architecture important?

- enables everyone to better understand the system
  - separation of concerns
    - computation from communication
    - architecture from implementation
- allows people to work on individual pieces of the system in isolation
  - explicit system structure (divide and conquer)
- prepares for extension of the system
- facilitates reuse and reusability, reduces development costs
  - system families, software product lines
  - component-based development

Components and connectors

Standard architectural framework:

- components are connected by connectors
- building blocks with which an architecture can be described
- no standard notation has emerged yet

Types of components

- **computational**: does a computation of some sort
  - e.g., function, filter
- **memory**: maintains a collection of persistent data
  - e.g., data base, file system, symbol table
- **manager**: contains state + operations, state is retained between invocations of operations
  - e.g., ADT, server
- **controller**: governs time sequence of events
  - e.g., control module, scheduler

Types of connectors

- procedure call (including RPC)
- data flow
  - e.g., pipes
- implicit invocation
  - e.g., interrupts
- message passing
- shared data
  - e.g., blackboard or shared data base
- instantiation

Terminology

- **component**
  - functionality, class, system, subsystem, legacy system, client, server, filter ...
- **connector**
  - communication, dependency, relationship, link, call, interaction, ...
- **style**
  - configuration, topology, pattern, composition rules, form, ...

Different terms with similar meaning – but many subtleties

Architectural styles

- recurring organizational patterns and idioms
  - established, shared understanding of common design forms
  - every mature engineering field has architectures
- abstraction of recurring composition and interaction characteristics in a set of architectures
**Style #1: Call-and-return**

**Problem:**
- hierarchy of functions
  - result of functional decomposition
- (usually) single thread of control
- context: language with nested procedures

**Solution:**

![Diagram](image1.png)

**Style #1: Call-and-return**

**Components:**
- functions / methods
  - usually single-threaded

**Connectors:**
- procedure calls
  - usually synchronously
- shared memory

**Topology:**
- hierarchy of (nested) functions
- interaction topologies can vary

**Style #1: Call-and-return**

**Strengths:**
- can change implementation without affecting clients
- can break problems into interacting agents
  - distributed across multiple machines / networks

**Weaknesses:**
- components must know their interaction partners
- topology hardwired
  - when partner changes, objects that explicitly invoke it must change
- indirect side effects
  - if A and B both use C, A’s effects on C can surprise B

**Style #2: Multi-layered architecture**

**Problem:**
- distinct, hierarchical classes of services
- each layer acts as a ...
  - service provider to layers “above”
  - service consumer from layers “below”
- operating systems, product families, ...

**Solution:**

![Diagram](image2.png)

**Style #2: Multi-layered architecture**

**Components:**
- services

**Connectors:**
- (typically) procedure calls
- API protocol

**Topology:**
- nested
- interaction topologies can be
  - **opaque**: layer $n+1$ can only call layer $n$
  - **translucent**: layer $n+1$ can call every layer $m \leq n$
- **virtual machine** style if layers are fully opaque

**Style #2: Multi-layered architecture**

**Strengths:**
- can increase levels of abstraction
  - can partition complex problems along the layers
- low coupling
  - especially for opaque layers
- supports reuse
  - implementation of a level can be swapped

**Weaknesses:**
- performance can suffer
  - opaque layers require communicating down through several layers
**Style #2: Multi-layered architecture**

**Variant:** protocol stacks
- complementary layered architectures on sender and receiver side
- standard architecture for networks

![Layered Architecture Diagram](image)

**Examples:**
- Application program
- User account management
- Kernel (binding processes and swapping)
- Dealing with application protocols
- Dealing with connections
- Dealing with packets
- Encapsulating and routing

**Style #3: Pipe-and-filter**

**Problem:**
- independent components solve simple tasks
- each component reads input in simple format, transforms, and writes output in simple format
- data transfer typically handled by OS
- want to glue components together to build system

**Solution:**

```
ls -> tee -> post -> lpt
file -> piped -> wc
```

**Components:**
- independent programs
- little local context used, no state maintained between instantiations

**Connectors:**
- pipes (i.e., queues)
  - data transfer typically incrementally (compare to sequential *batch processing*)
- data transferred in pure ASCII or XML format

**Topology:**
- data flow graph

**Strengths:**
- supports reuse
  - filters only need to agree on the data format
- architecture can easily be reconfigured

**Weaknesses:**
- sharing global data is expensive or limiting
- can be difficult to design incremental filters
  - not appropriate for interactive applications
- error handling
  - e.g., some intermediate filter crashes
- untyped, low-level data format

**Example:**

![Pipe-and-Filter Diagram](image)
**Style #4: Repository architecture**

**Problem:**
- long-lived, richly structured, shared data
- to be manipulated in many different ways by different clients

**Solution:**

**Components:**
- central data repository + schema
  - can be active (send notifications)
- independent operators
  - interact with database by queries and updates

**Connectors:**
- messages / procedure calls
- shared data

**Topology:**
- star (components only communicate with repository)

**Style #4: Repository architecture**

**Strengths:**
- efficient way to share large amounts of data
- data integrity localized to repository module

**Weaknesses:**
- repository data model is compromise between clients
- schema evolution is difficult and expensive
- distribution can be a problem

**Example:**

**Style #5: Client/server**

**Problem:**
- some components are service provides, others users
- irregular usage patterns
- asymmetric relation: service requests driven by users (pull)
- context: distributed systems

**Solution:**

**Components:**
- clients: user-facing, little persistent state, active (request services)
- servers: “in the back office”, maintains persistent state and offers services, passive

**Connectors:**
- remote procedure calls or network protocols
- server doesn’t know identity of clients

**Topology:**
- clients surround the server(s)
Style #5: Client/server

**Strengths:**
- makes effective use of networked systems
- easy to add / upgrade servers
- redundancy relatively straightforward.

**Weaknesses:**
- communication may be expensive and slow
- denial of service attacks
- data interchange complicated, data integrity functionality must be implemented for each server

Example:

![Client/server architecture for film library](image)

Style #6: Peer-to-peer

**Variant** of client/server:
- each component acts both as server and client
- no centralized component
- flexible communication structure

**Strengths:**
- efficiency: all clients provide resources
- scalability – system capacity grows with number of clients
- robustness – data is replicated over peers – no single point of failure in the system (in pure peer-to-peer style)

**Weaknesses:**
- architectural complexity – more demanding of peers (compared to client-server).
- resources are distributed and not always available

Style #7: Event-based

**Problem:**
- loosely coupled collection of components
- application likely to be reconfigured
- context: requires event handler – through OS or language

**Solution:**

![Event-based components](image)
**Style #7: Event-based**

Example:

![Publish/Subscribe Architecture Diagram](image1)

- **Problem:**
  - separation of UI from application

- **Solution:**
  - separation of concerns
  - standard architecture

- **Weaknesses:**
  - can be too much overhead for small models or simple UIs

**Style #8: Model-view-controller**

- **Components:**
  - **Model:** holds data
  - **View:** draws visualization
  - **Controller:** manages interaction

- **Connectors:**
  - typically procedure calls

- **Topology:**
  - typically single-threaded

**Style #8: Model-view-controller**

- **Example:** MVC on the internet
  - **Model:** underlying system that manages the information.
  - **View:** generates the HTML code to be displayed by the browser
  - **Controller:** interprets HTTP POST transmissions coming back from the browser

**Design Patterns**
Design patterns

A design pattern is the outline of a reusable solution to a general software design problem encountered in a particular context.

Good patterns...
• are as general as possible
• solve recurring problems
• describe a proven and effective solution
  – for the problem in the indicated context

⇒ Studying patterns is an effective way to learn from the experience of others!

Pattern description

Patterns are typically described in the following style:
• Context: general situation where the pattern applies
• Problem: short description of the main difficulty
• Forces: issues to consider when solving the problem
• Solution: recommended way to solve the problem in the given context
  – typically a UML class diagram
• Antipatterns: solutions that are inferior or do not work in this context (optional)
• Related patterns (optional)
• References: who developed or inspired the pattern

Example: Abstraction-Occurrence

• Context: in a domain model you find a set of related objects (occurrences) such that the objects
  – share common information
  – but also differ from each other in important ways
• Problem: what is the best way to represent such occurrences in a class diagram?
• Forces: represent the occurrences without duplicating the common information
• Solution:

  ![UML diagram](image)

Example: Abstraction-Occurrence

• Antipatterns:

  ![UML diagram](image)

Example: General Hierarchy

• Context: objects in a hierarchy can have one or more objects above (superiors) and below (subordinates) them, but some objects cannot have any subordinates
• Problem: how do you represent an object hierarchy where some objects cannot have subordinates?
• Forces:
  – you want a flexible way to represent the hierarchy
  – the objects have many common properties and operations
**Example: General Hierarchy**

- **Solution:**
  ![General Hierarchy Solution Diagram]

- **Antipattern:**
  ![General Hierarchy Antipattern Diagram]

**The Gang of Four (GoF)**

- patterns originated in the mid-70’s as architectural concept *(C. Alexander, A Pattern Language)*
- first application to programming in the mid-80’s *(K. Beck and W. Cunningham)*
- became wildly popular in computer science in 1995 *(E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software)*
- known informally as the “Gang of Four” (GoF)

**GoF pattern description template**

- **Pattern Name and Classification:** descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:** description of the goal behind the pattern and the reason for using it.
- **Also Known As:** other names for the pattern.
- **Motivation:** scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:** situations in which this pattern is usable; the context for the pattern.
- **Structure:** graphical representation of the pattern: class diagrams and interaction diagrams may be used for this purpose.
- **Participants:** listing of the classes and objects used in the pattern and their role in the design.
- **Collaboration:** description of how classes and objects used in the pattern interact with each other.
- **Consequences:** description of the results, side effects, and trade-offs caused by using the pattern.
- **Implementation:** description of an implementation of the pattern; solution part of the pattern.
- **Sample Code:** illustration of how the pattern can be used in a programming language.
- **Known Uses:** examples of real usages of the pattern.
- **Related Patterns:** other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

**GoF patterns**

- **Creational patterns** for class instantiation
- **Structural patterns** for class & object composition
  - use inheritance to
    - compose interfaces
    - define ways to obtain new functionality via composition
  - “wrap” classes to
    - modify interface
    - extend functionality
    - restrict access
- **Behavioural patterns** for object communications
  - especially useful when using multiple abstractions

**GoF patterns**

- **By Purpose**
  - **Creational:** Factory Method, Adapter (Clone), Abstract Factory, Builder, Prototype, Singleton
  - **Structural:** Adapter (object), Bridge, Decorator, Facade, Proxy, Chain of Responsibility, Command, Interpreter, Mediator, Memento, Observer, State, Strategy, Visitor

---

**Example: General Hierarchy**

- **Solution:**
  ![General Hierarchy Solution Diagram]

- **Antipattern:**
  ![General Hierarchy Antipattern Diagram]

**The Gang of Four (GoF)**

- patterns originated in the mid-70’s as architectural concept *(C. Alexander, A Pattern Language)*
- first application to programming in the mid-80’s *(K. Beck and W. Cunningham)*
- became wildly popular in computer science in 1995 *(E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software)*
- known informally as the “Gang of Four” (GoF)

**GoF pattern description template**

- **Pattern Name and Classification:** descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:** description of the goal behind the pattern and the reason for using it.
- **Also Known As:** other names for the pattern.
- **Motivation:** scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:** situations in which this pattern is usable; the context for the pattern.
- **Structure:** graphical representation of the pattern: class diagrams and interaction diagrams may be used for this purpose.
- **Participants:** listing of the classes and objects used in the pattern and their role in the design.
- **Collaboration:** description of how classes and objects used in the pattern interact with each other.
- **Consequences:** description of the results, side effects, and trade-offs caused by using the pattern.
- **Implementation:** description of an implementation of the pattern; solution part of the pattern.
- **Sample Code:** illustration of how the pattern can be used in a programming language.
- **Known Uses:** examples of real usages of the pattern.
- **Related Patterns:** other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

**GoF patterns**

- **Creational patterns** for class instantiation
- **Structural patterns** for class & object composition
  - use inheritance to
    - compose interfaces
    - define ways to obtain new functionality via composition
  - “wrap” classes to
    - modify interface
    - extend functionality
    - restrict access
- **Behavioural patterns** for object communications
  - especially useful when using multiple abstractions
GoF patterns

Creational patterns

Singleton
- **Context:** it is very common to find classes for which only one instance should exist
- **Problem:** how do you ensure that it is never possible to create more than one instance of a singleton class?
- **Forces:**
  - the use of a public constructor cannot guarantee that no more than one instance will be created
  - the singleton instance must also be accessible to all classes that require it

Factory Method and Abstract Factory
- **Creational patterns** abstract the object instantiation process (i.e., `new`)
  - hide how objects are created
  - help make the overall system independent of how its objects are created and composed
- **Class creational patterns** focus on the use of inheritance to decide the object to be instantiated
  - **Factory Method**
- **Object creational patterns** focus on the delegation of the instantiation to another object
  - **Abstract Factory**

Singleton
- **Solution:**

```java
<Singleton>
    theInstance
    getInstance()

Company
    theCompany
    Company() {private
    getInstance() --
    if (theCompany==null)
        theCompany= new Company();
    return theCompany;
```

Maze example
Consider a maze game with the following classes:
Maze example

```java
class MazeGame {
    public Maze makeMaze() // factory method
        return new Maze();
    public Wall makeWall(int n) // factory method
        return new Wall(n);
    public Room makeRoom(int n) // factory method
        return new Room(n);
}
```

Maze example

```java
class EnchantedMazeGame extends MazeGame {
    public Maze makeMaze() // factory method
        return new EnchantedMaze();
    public Wall makeWall() // factory method
        return new EnchantedWall();
    public Room makeRoom() // factory method
        return new EnchantedRoom();
}
```

Maze example

```java
Maze makeMaze(int n) {return new Room(n);}
```

Maze example

```java
Maze makeMaze() {
    Maze maze = makeMaze();
    Room room = makeRoom();
    Wall wall = makeWall();
    Door door = makeDoor();
    return maze;
}
```

Maze example

```java
Door door = makeDoor(r1, r2);
```

Maze example

```java
Maze makeMaze() {return new Room(n);}
```

Maze example

```java
Maze makeMaze() {return new Room(n);}
```

Maze example

```java
Maze makeMaze() {return new Room(n);}
```

Maze example

```java
Maze makeMaze() {return new Room(n);}
```

Structural patterns

- Proxy
  - Provide a surrogate or placeholder for another object to control access to it.
- Bridge
  - Decouple abstraction from implementation so they can vary independently.
- Flyweight
  - Use sharing to support large numbers of fine-grained objects.
- Decorator
  - Attach additional responsibilities to an object dynamically. Enhance or implement new functionality.
- Facade
  - 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.
- Composite
  - Composite objects into tree structures to represent part-whole hierarchies. Let clients treat individual objects and compositions of objects uniformly.

GoF original patterns

- Adapter
  - Context:
    - you are building an inheritance hierarchy and want to incorporate it into an existing class.
    - the reused class is also often already part of its own inheritance hierarchy
  - Problem: how to obtain the power of polymorphism when reusing a class whose methods have the same function but not the same signature as the other methods in the hierarchy?
  - Forces: you do not have access to multiple inheritance or you do not want to use it
**Behavioral patterns**

- **Iterator**: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Blackboard**: Generalized observer, which allows multiple readers and writers. Communicates information system-wide.
- **State**: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Specification**: Reimplementable business logic in a framework fashion.
- **Restorer**: An alternative to the existing Memento pattern.
- **Null Object**: Designed to act as a default value of an object.
- **Memento**: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Interpreter**: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Command**: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

**GoF original patterns**

**Behavioral patterns**

- **Visitor**: Represent an operation to be performed on elements of an object structure. Lets you define new operation without changing the classes of the elements on which it operates.
- **Observer or Publish/Subscribe**: Define: a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Strategy**: Define a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Chain of responsibility**: Avoid coupling sender of a request to receiver by giving the request to a chain of handlers. Chain re- tains objects and passes request along chain until an object handles it.
- **Template method**: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

**GoF original patterns**