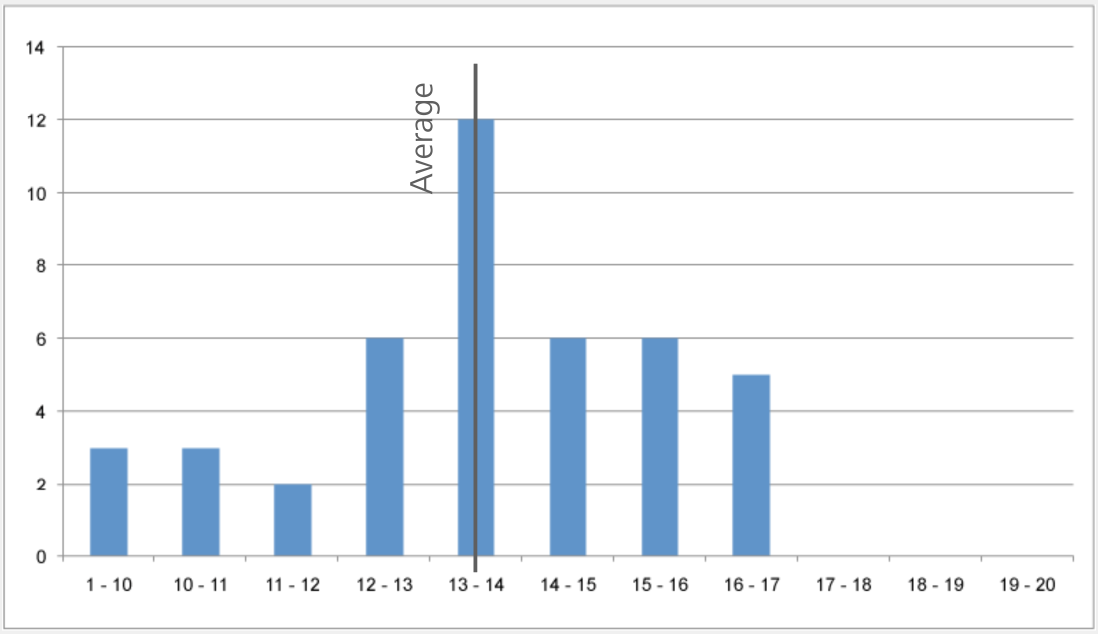
Discussion SE Exercise 4

Dustin Wüest and Cédric Jeanneret

Requirements Engineering Research Group
Department of Informatics
University of Zurich

SE Exercise 4 Results



Ex 2.1: Architecture Styles

1. What are the structural elements of the systems (their roles, their responsibilities)?
2. What is the nature of communication between these elements?

Find an architectural style that is appropriate.

Ex 2.1.A: Secretary
Pipes and Filters

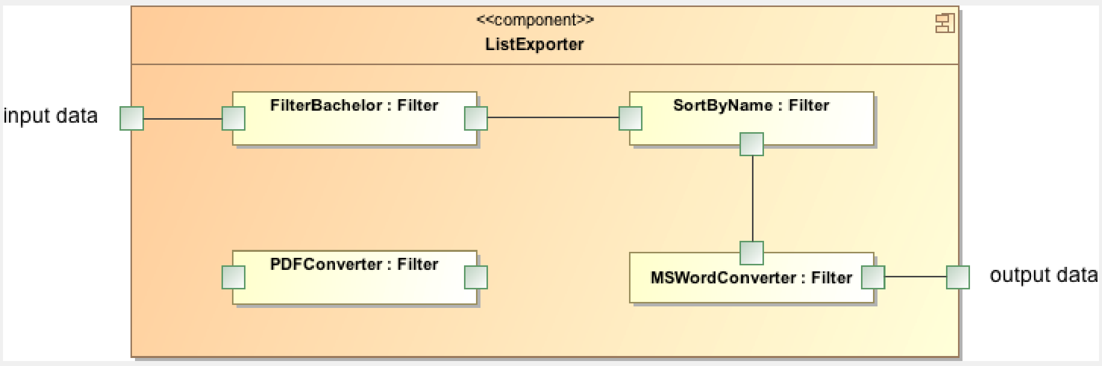
Components
- Filters: Sorters, Filters, Converters

Connector
- Pipes

Advantages
- No complex components interactions
- Reusability and maintainable (add, arrange and substitute filter)

Disadvantages
- Computational overhead (e.g. parsing in several filter)
- Batch processing

Ex 2.1.A: Secretary
Pipes and Filters



Ex 2.1.B: Students
Publish / Subscribe

Components
- Publishers / Subscribers

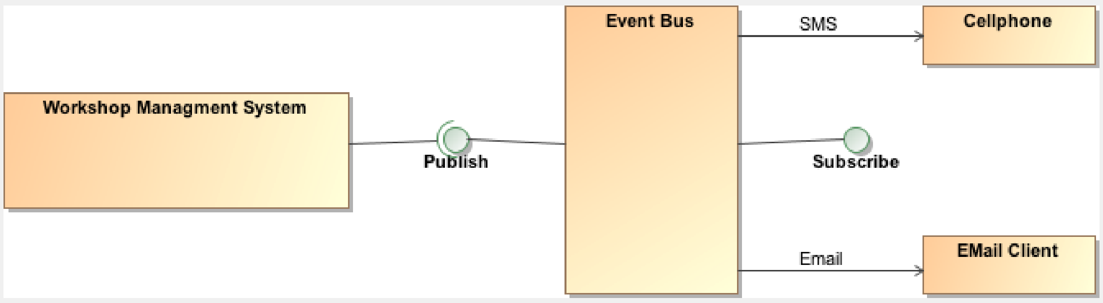
Connectors
- Message distribution and registration infrastructure.

Announcers of events do not know which components will be affected by those events.

Advantages
- Very low-coupling of components

Disadvantages
- Reliability (no guarantee of getting an answer)
- Simplicity (no control over the order of answers)
- Understandability (difficulty for reasoning independently of subscribers)
- Event abstraction does not cleanly lend itself to data exchange

Ex 2.1.B: Students
Publish / Subscribe



Ex 2.1.C: Department's head
Virtual Machine (Interpreter)

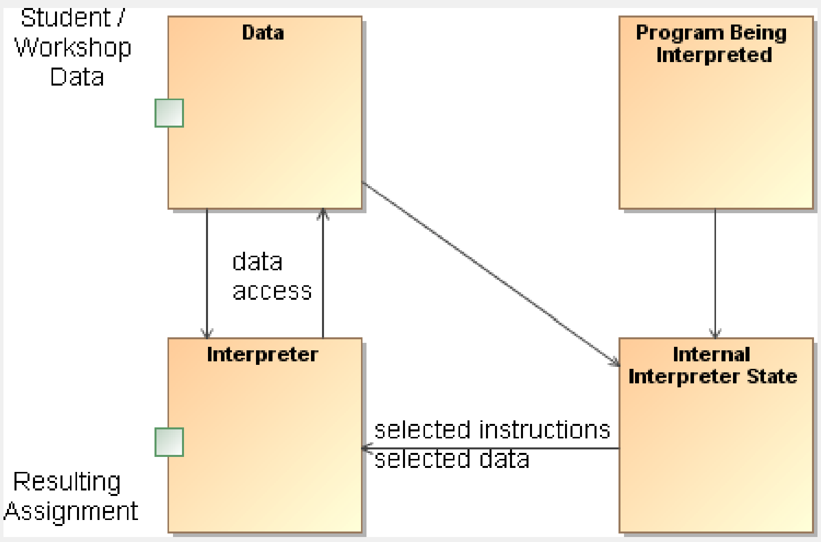
Components:
- Command interpreter
- Program state (data)
- Interpreter internal state
- Program to be interpreted

Connectors:
- Procedure calls / Memory access (very tightly coupled)

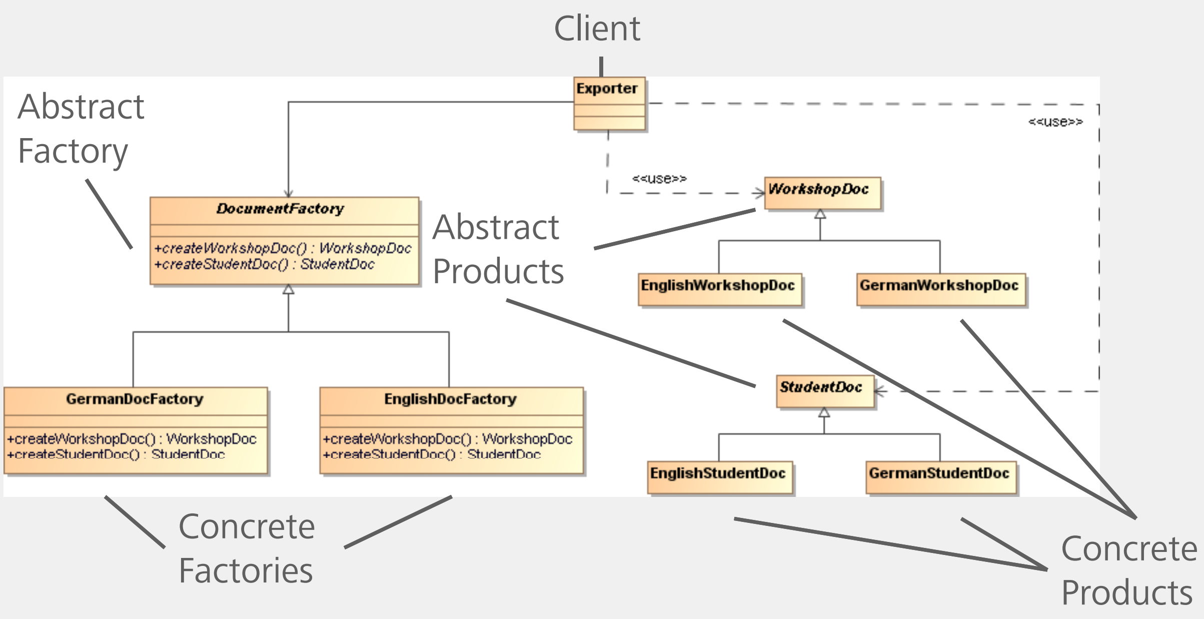
Advantages:
- Highly dynamic behavior possible
- End-user programmability

Disadvantages:
- Slow performance due to interpretation
- Difficulties for maintenance

Ex 2.1.C: Department’s head
Virtual Machine (Interpreter)



Ex 2.2.A: Creational Patterns
Abstract Factory



Ex 2.2.A: Creational Patterns
Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Uses

- Isolate clients from implementation classes
- A system should be independent of how its products are created, composed and represented
- Configuration of a system with one of multiple families of product
- Ensure consistency among products

Consequences

- Supporting new kind of products is difficult

Ex 2.2.A: Creational Patterns
Factory Method?

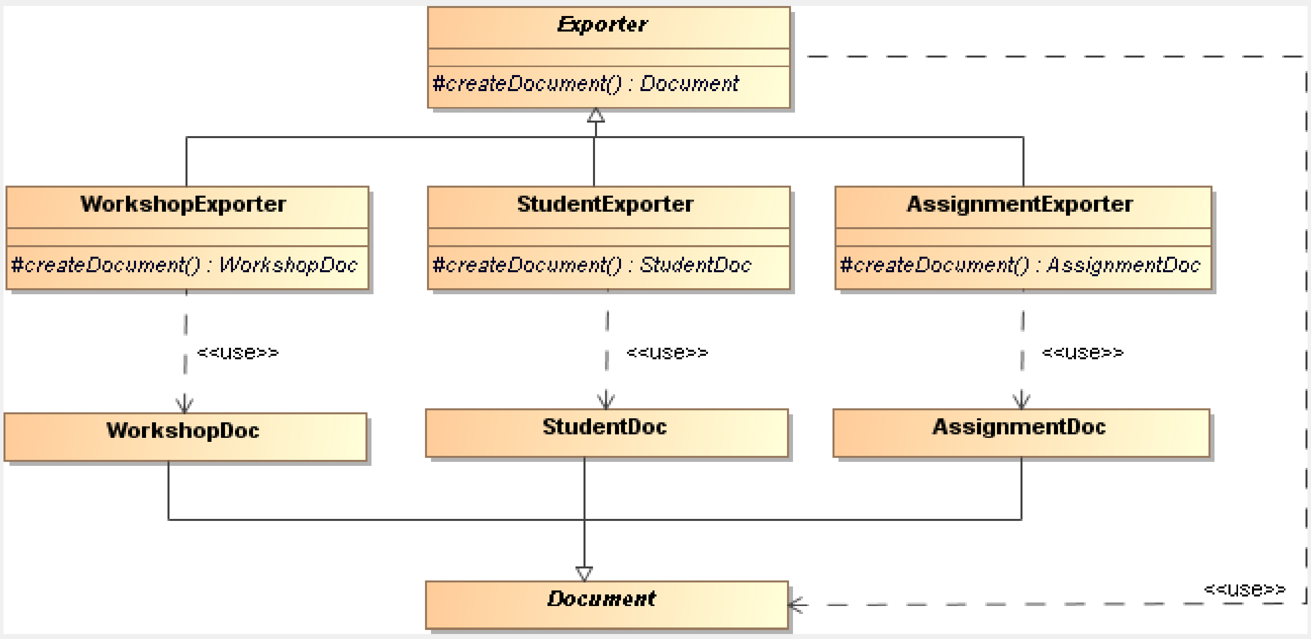
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Uses

- A class wants its subclasses to specify the objects it creates
- Connects parallel class hierarchies (e.g.: Figures and Manipulators)

University of Zurich

Ex 2.2.A: Creational Patterns
Factory Method?



University of Zurich

Ex 2.2.A: Creational Patterns
Factory Method?

		Languages →		
		German	*English*	…
Documents →	*WorkshopDoc*			
	StudentDoc			
	AssignmentDoc			
	…			

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. (GoF, p136)

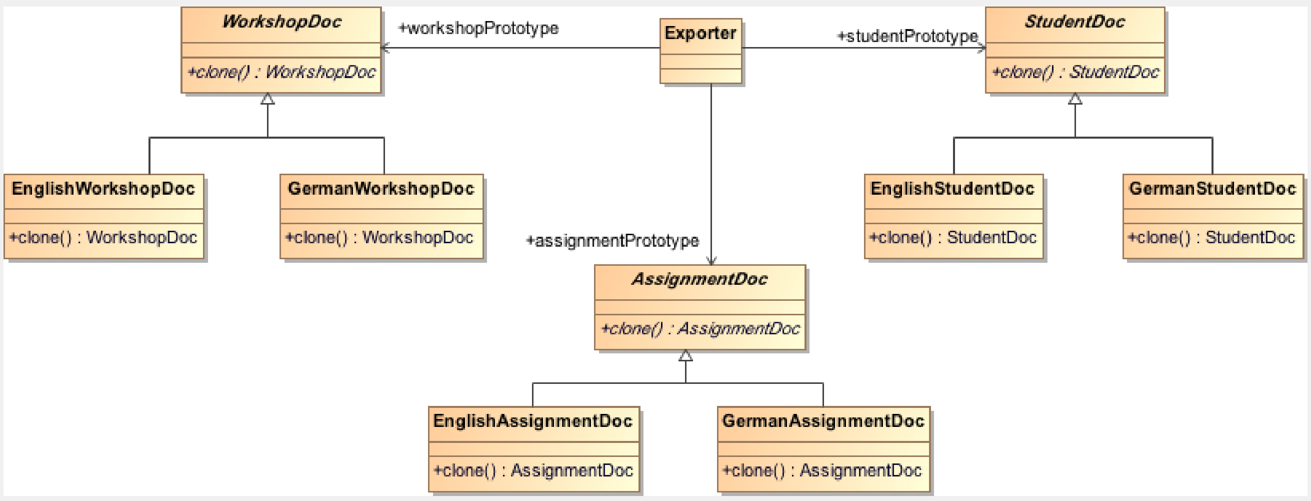
Ex 2.2.A: Creational Patterns *Prototype*

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Uses

- A system should be independent of how its products are created, composed and represented
  - The classes to instantiate are specified at run-time
  - To avoid building a hierarchy of factories

Ex 2.2.A: Creational Patterns *Prototype*



Ex 2.2.A: Creational Patterns *Builder*

University of Zurich

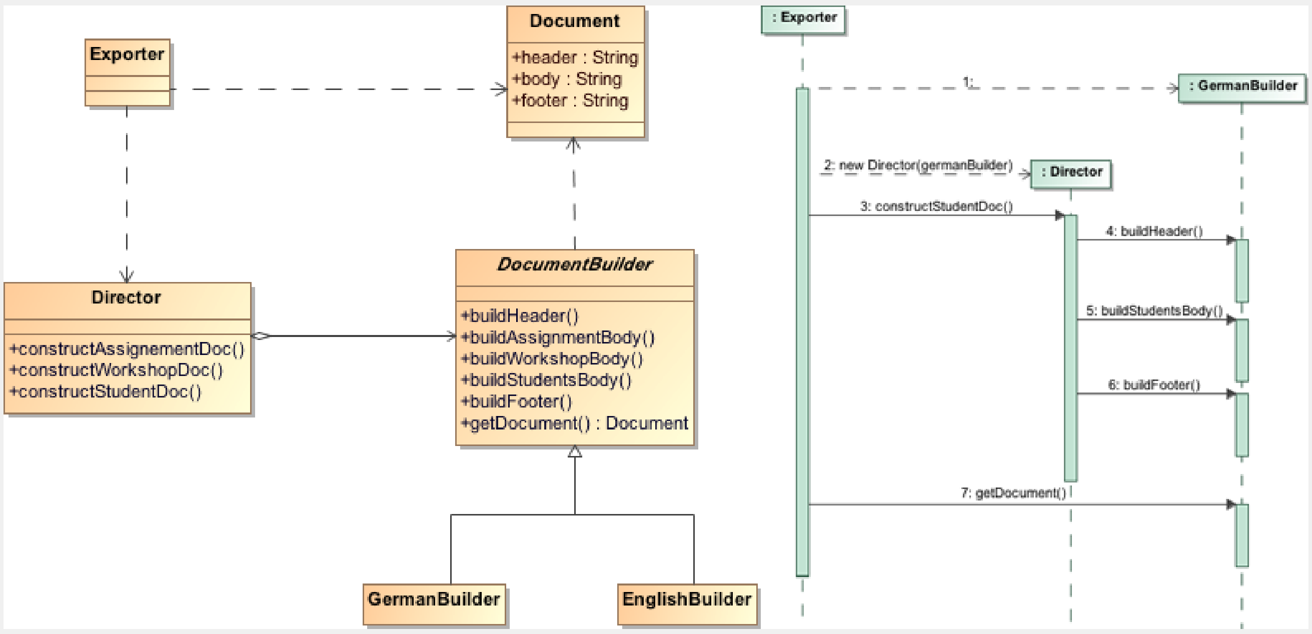
Separate the construction of a ***complex*** *object from its representation so that the same construction process can create different representations.*

Consequences

- It isolates code for construction and representation
- It provides a finer control over the construction process

Ex 2.2.A: Creational Patterns *Builder*

University of Zurich



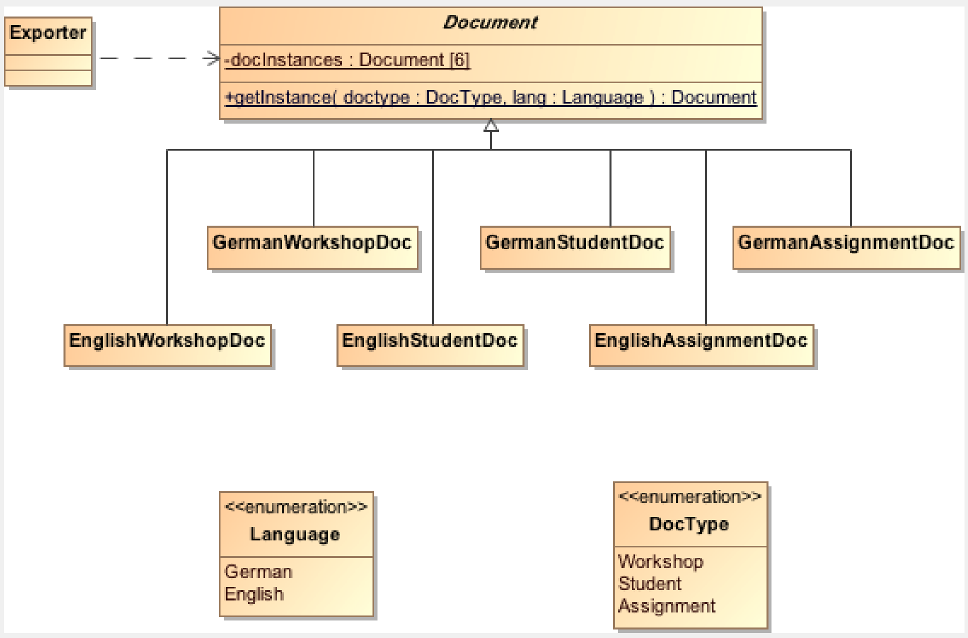
Ex 2.2.A: Creational Patterns
Singleton

*Ensure a class **only** has **one** instance, and provide a global point of access to it.*

Uses:

- There must be exactly one instance of a class and it must be accessible to clients from a well-know access point
- The sole instance should be able to use an extended instance without modifying their code

Ex 2.2.A: Creational Patterns
Singleton



Ex 2.2.A: Creational Patterns *Summary*

University of Zurich

Instead of creating documents itself:

Document doc = new EnglishWorkshopDoc();

An exporter object can:

- Call an abstract method. This method is implemented in a subclass which will decide which class to instantiate (*FactoryMethod*)
- Delegate the creation of document to an object (*AbstractFactory*)
- Use a object that will build a document under its direction (*Builder*)
- Clone a prototype (*Prototype*)

Ex 2.3: Creational Patterns in Java

University of Zurich

Singleton:

- java.lang.Runtime

Abstract Factory

- javax.xml.validation.SchemaFactory (creates a Schema)

Factory Method:

- java.net.SocketImpl (creates a java.io.InputStream)

Prototype:

- java.lang.Cloneable

Builder:

- java.lang.StringBuilder (builds String)

Ex 2.2.B: Structural Pattern *Proxy*

Provide a surrogate or placeholder for another object to control access to it.

Uses

- Remote Proxy (local representative for a remote object)
- Protection Proxy (verification of access rights)
- Virtual Proxy (creation of an object on demand)
- Smart References (housekeeping, locking)

Ex 2.2.B: Structural Pattern *Adapter*

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

A proxy has the same interface than its subject.

An adapter adapts the interface of its adaptee to a target.

Ex 2.2.B: Structural Pattern *Facade*

Facade provides a unified interface to ***a set of interfaces*** *in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.*

Facade defines a new interface whereas adapter reuses an old interface.

Ex 2.2.B: Structural Pattern *Bridge*

Bridge decouples an ***abstraction*** *from its* ***implementation*** *so that the two can vary independently.*

Adapter is usually applied to objects after they have been designed, Bridge define a stable interface for various (future) implementations.

Ex 2.2.B: Structural Pattern *Decorator*

A decorator attaches additional responsibilities to an object dynamically ***keeping the same interface****. Decorators provide a flexible alternative to subclassing for extending functionality.*

Proxy provides (or prevent) access to its subject (which provides the key functionality), decorator completes the functionalities of a component (which only provides part of the functionality)

Ex 2.3: Structural Patterns in Java

Adapter:
- java.io.InputStreamReader (adapts a InputStream to a Reader)
- ~~java.awt.event.MouseAdapter~~ (doesn’t adapt anything)

Decorator:
- java.io.FilteredInputStream (decorates an InputStream)
- javax.swing.border.TitledBorder (decorates a Border)

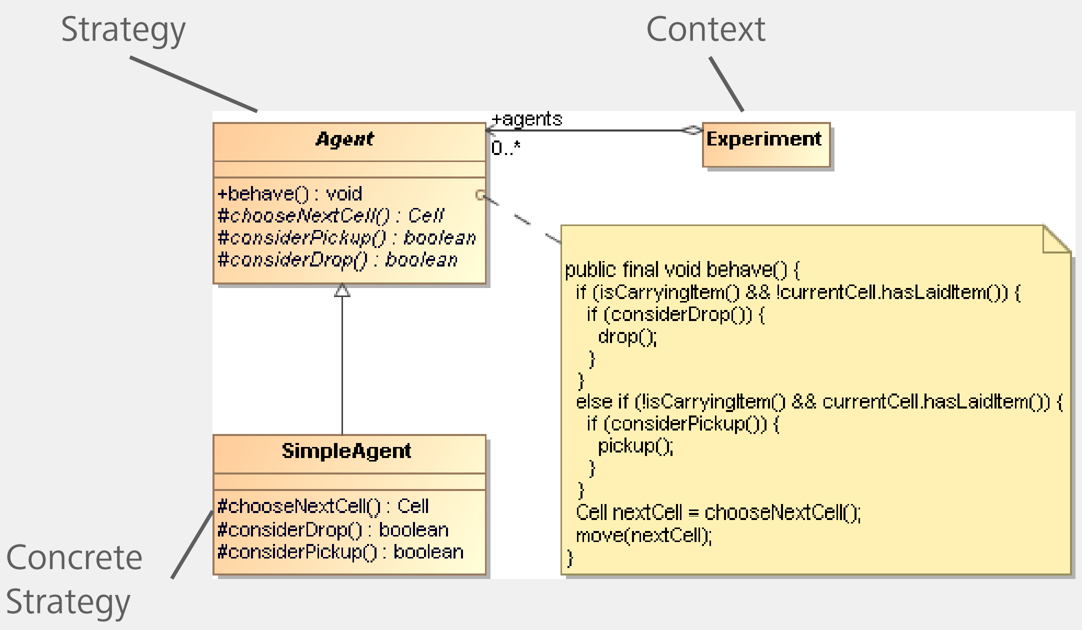
Composite:
- java.awt.Component (Container, like Panel, contains Label, Button)
- ~~java.awt.Composite~~ (refers to “compositing”, esp. “alpha compositing”)

Bridge:
- java.net.Socket (implemented by a SocketImpl)

Facade:
- java.awt.Font

Proxy:
- java.lang.reflect.Proxy

Ex 2.2.C: Behavioral Patterns *Strategy*



```
public final void behave() {
  if (isCarryingItem() && !currentCell.hasLaidItem()) {
    if (considerDrop()) {
      drop();
    }
  }
  else if (!isCarryingItem() && currentCell.hasLaidItem()) {
    if (considerPickup()) {
      pickup();
    }
  }
  Cell nextCell = chooseNextCell();
  move(nextCell);
}
```

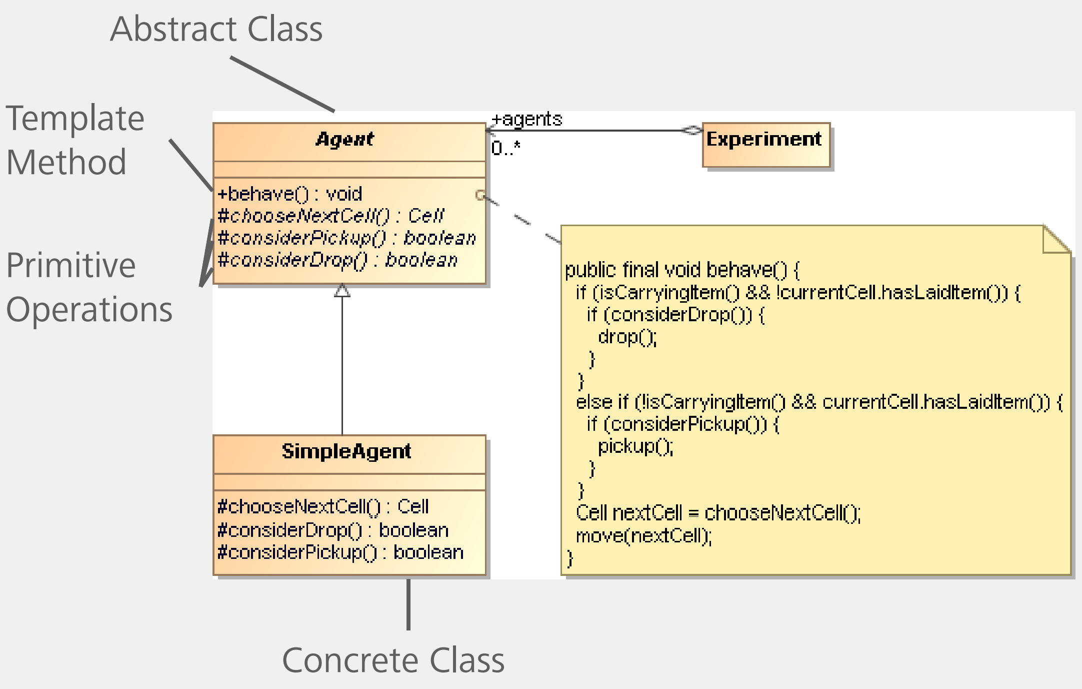
Ex 2.2.C: Behavioral Patterns *Strategy*

Strategy defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Uses:

- Configure a class with one of many behaviors (e.g.: ciphers, like IDEA or AES)
- Different variants of an algorithm with various space/time tradeoff (e.g.: sort algorithms)
- Hide complex algorithm-specific data structure to a client
- Remove many conditional statements in a class operations

Ex 2.2.C: Behavioral Patterns *Template Method*



```
public final void behave() {
  if (isCarryingItem() && !currentCell.hasLaidItem()) {
    if (considerDrop()) {
      drop();
    }
  }
  else if (!isCarryingItem() && currentCell.hasLaidItem()) {
    if (considerPickup()) {
      pickup();
    }
  }
  Cell nextCell = chooseNextCell();
  move(nextCell);
}
```

Ex 2.2.C: Behavioral Patterns *Template Method*

A template method defines 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.

Uses:

- Control subclass extensions by defining “hooks” operation
- Implement the invariant part of an algorithm and let subclasses implements the parts that can vary
- Avoid code duplication (of common behavior among several classes)

University of Zurich

Ex 2.2.C: Behavioral Patterns *State*

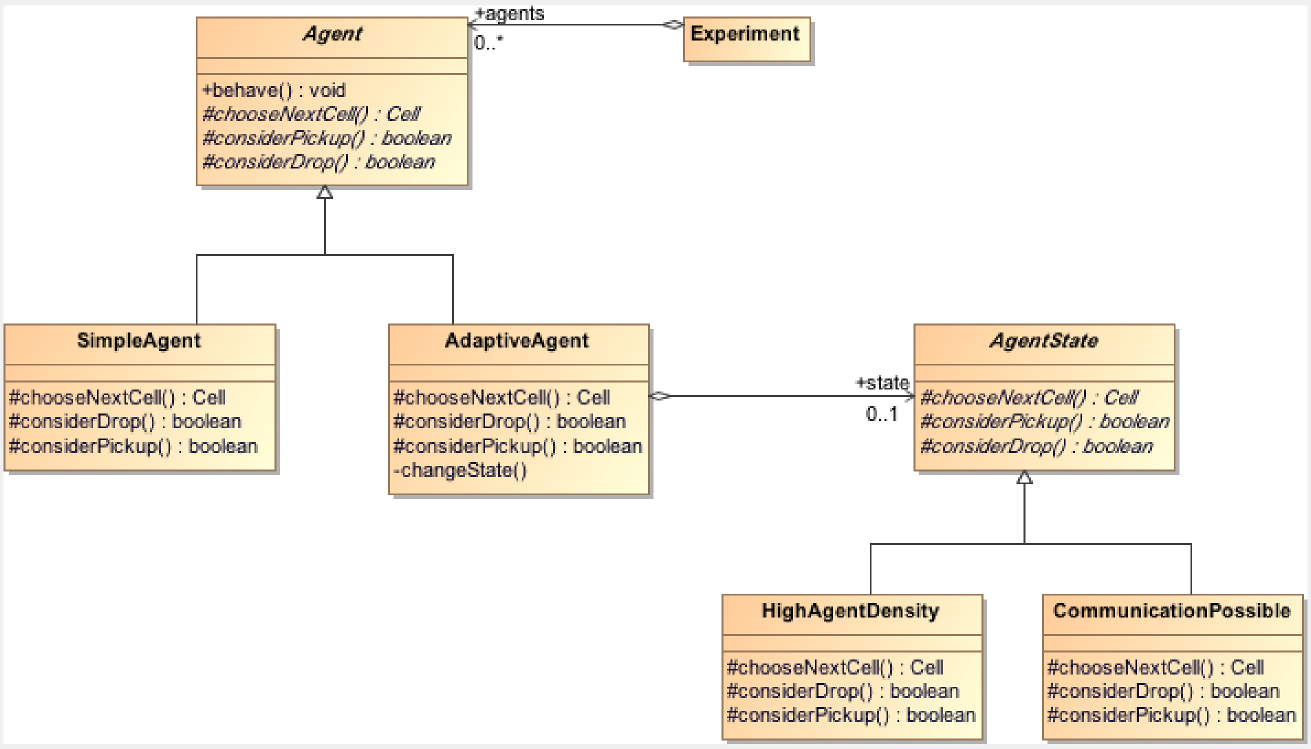
State allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Uses:

- An object's behavior depends on its state and it must change its behavior at run-time depending on state
- Operations have large, multipart conditional statements that depend on the object's state

University of Zurich

Ex 2.2.C: Behavioral Patterns *State*


Agent
+behave() : void
#chooseNextCell() : Cell
#considerPickup() : boolean
#considerDrop() : boolean
+agents
0..*
Experiment
SimpleAgent
#chooseNextCell() : Cell
#considerDrop() : boolean
#considerPickup() : boolean
AdaptiveAgent
#chooseNextCell() : Cell
#considerDrop() : boolean
#considerPickup() : boolean
-changeState()
+state
0..1
AgentState
#chooseNextCell() : Cell
#considerPickup() : boolean
#considerDrop() : boolean
HighAgentDensity
#chooseNextCell() : Cell
#considerDrop() : boolean
#considerPickup() : boolean
CommunicationPossible
#chooseNextCell() : Cell
#considerDrop() : boolean
#considerPickup() : boolean

Ex 2.2.C: Behavioral Patterns *Mediator*

University of Zurich

A mediator is an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

In our case, a mediator would act like a “chief agent”, organizing and managing his group. This is absolutely contrary to the principle of the framework!

Ex 2.2.C: Behavioral Patterns *Summary*

University of Zurich

Encapsulate variation

- Strategy encapsulates an algorithm
- State encapsulates a state-dependant behavior
- Mediator encapsulates the protocol between objects
- Iterator encapsulates the way an aggregate object is traversed
- Command encapsulates a request to a client
- Visitor encapsulates an operation on a object structure

Decouple Sender/Receivers with various trade-off

- Command, Observer, Mediator, Chain of Responsibility

Ex 2.3: Behavioral Patterns in Java

University of Zurich

Iterator:
- java.util.Iterator (iterates over a Collection)

Observer:
- java.util.Observer (observes an Observable)

Strategy:
- java.awt.LayoutManager (e.g.: GridBagLayout, FlowLayout,…)

Template Method:
- java.io.InputStream (read(byte[] b) uses the abstract primitive operation read())

Visitor:
- javax.lang.model.element.ElementVisitor (visits a Java 6 AST, made of Element)

Command:
- javax.activation.CommandObject

Mediator:
- java.awt.KeyboardFocusManager (manages the “focus” among javax.swing.JComponent)