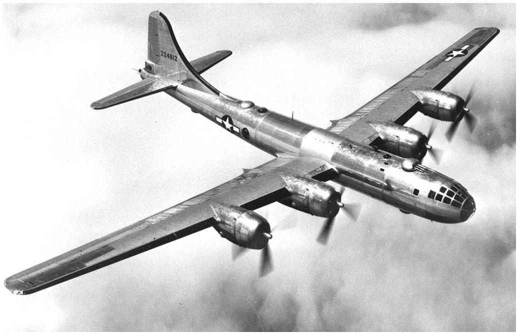
Reverse Engineering

Reverse Engineering

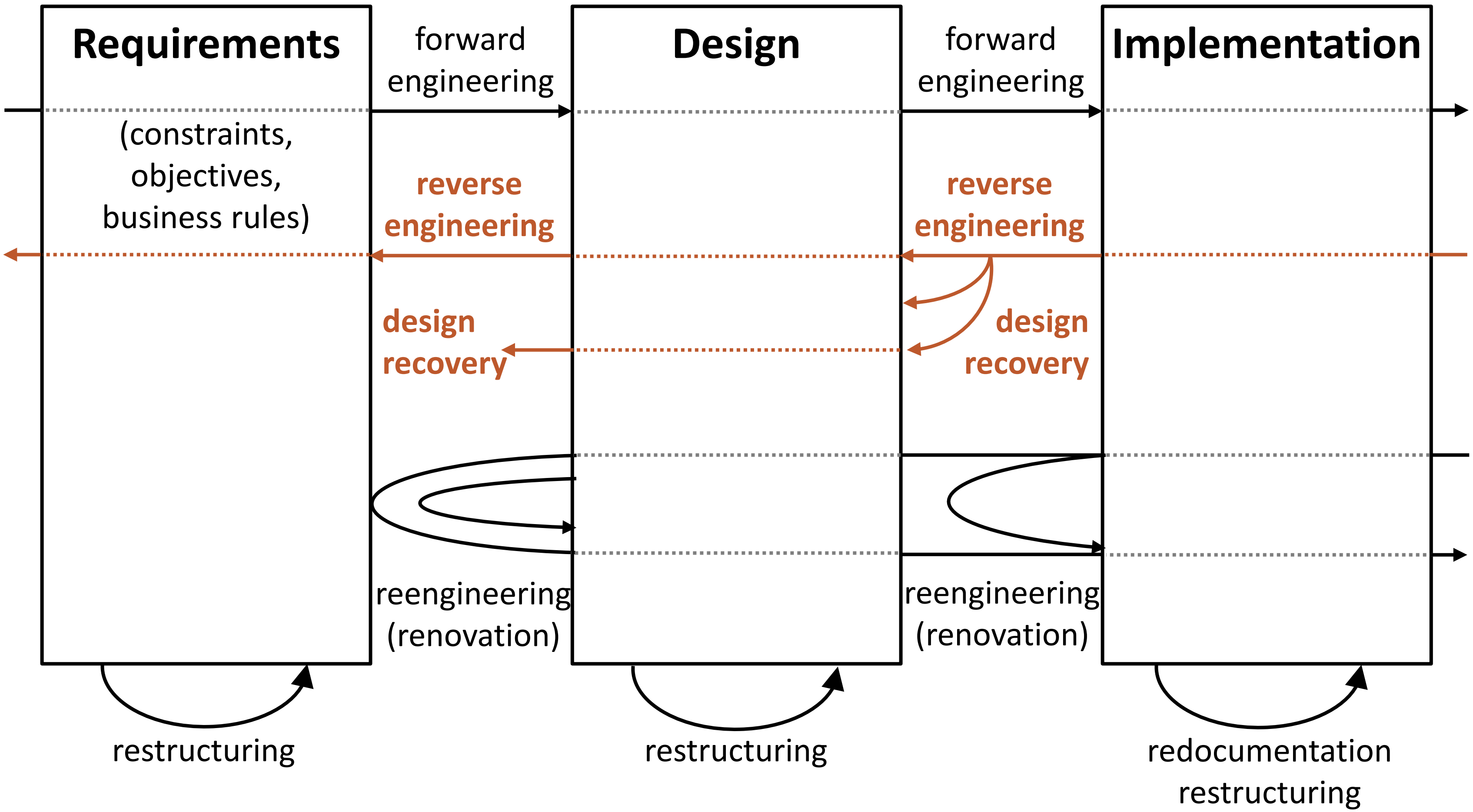
- also called back engineering
- process of :
  1. extracting the knowledge or design information from anything man-made
  2. reproducing it or anything based on the extracted information

Reverse Engineering in Reality

- World War 2 in 1944, three B-29 Bombers with emergencies landed in Vladivostock, Russia
  - long range bomber designed to reach over the Pacific
- Stalin steals and decides to make a bolt to bolt exact copy – TU-4 NATO (a.k.a BULL)
  - it would take 5 years for Russian to build from scratch
- disassemble, analyse & measure, copy, test, run
  - 1disassembled, 1referece model, 1 pilot training



Reverse Engineering



Chikofsky, Elliot J., and James H. Cross. "Reverse engineering and design recovery: A taxonomy." *IEEE software* 7.1 (1990): 13-17.

Forward vs. Reverse Engineering

- forward engineering
  - traditional process of moving from high-level abstractions and logical, implementation-independent design to the physical implementation of a system

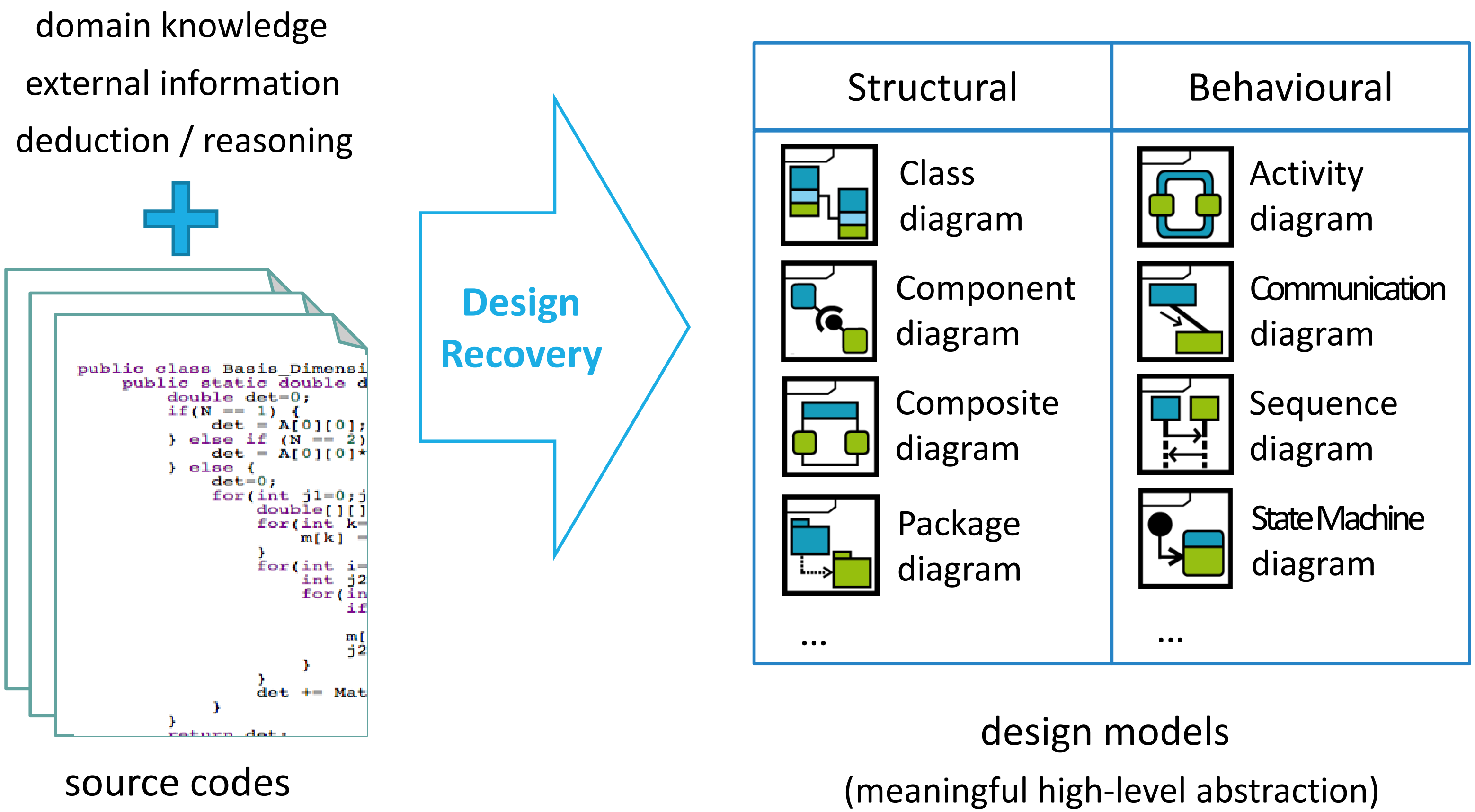
- reverse engineering
  - process of analysing a subject system to :
    1. identify the system's components and their interrelationships
    2. create representations of the system at a higher level of abstraction

Redocumentation

- creation or revision of a semantically equivalent representation
  - e.g., dataflow, data structure, control flow, …
- common tool supports
  - provides easier ways to visualise relationships among system components
    - pretty printers
    - diagram generators
    - cross-reference listing generators

Design Recovery

- subset of reverse engineering



Restructuring

- transformation from one representation form to another while preserving external behaviour
  - e.g., altering code to improve its structure
- transformation, recasting, reshaping of data models, design plan, requirements structures
  - e.g., normalization
- can be performed with a knowledge of structural form BUT without an understanding of meaning
  - NOT including modification w.r.t new requirements

Reengineering

- also known as renovation / reclamation
- examination and alteration of a system to reconstitute it in a new form and the subsequent implementation of the new form
- involves reverse engineering followed by forward engineering or restructuring
  - may include modification w.r.t new requirements

Objectives of Reverse Engineering

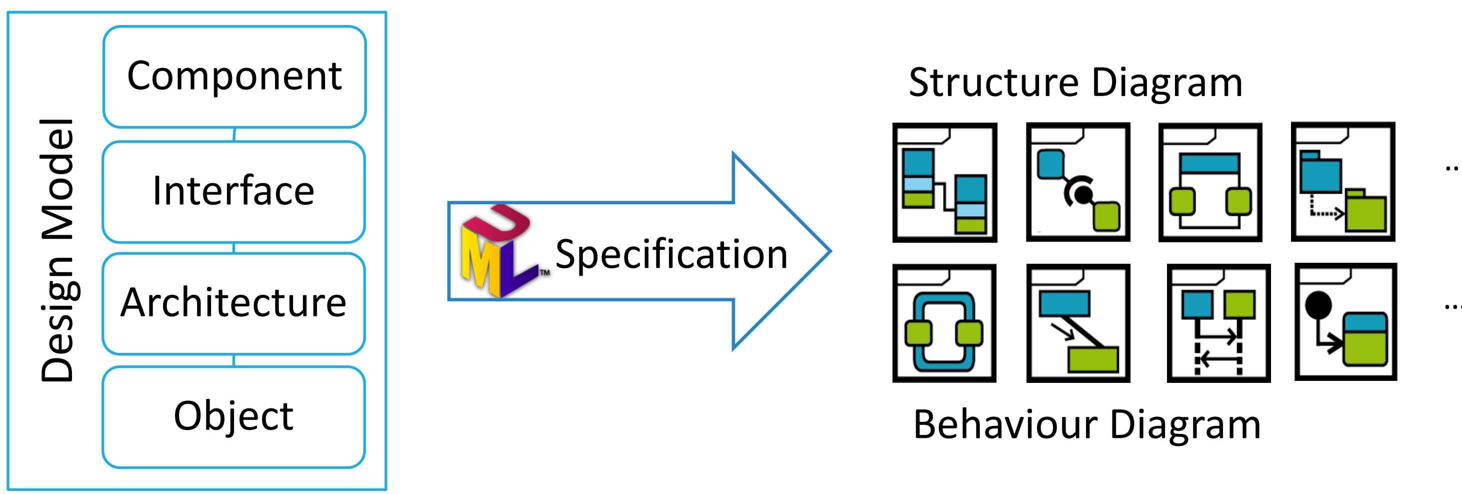
- to increase overall comprehensibility of a software system for both maintenance and new development

- key objectives :
  - cope with complexity
  - generate alternative views
  - recover lost information
  - detect side effects
  - synthesise higher abstraction
  - facilitate reuse

Software Design

Software Design

- process of creating a specification of a software artefact, intended to accomplish goals, using a set of primitive components and subject to constraints



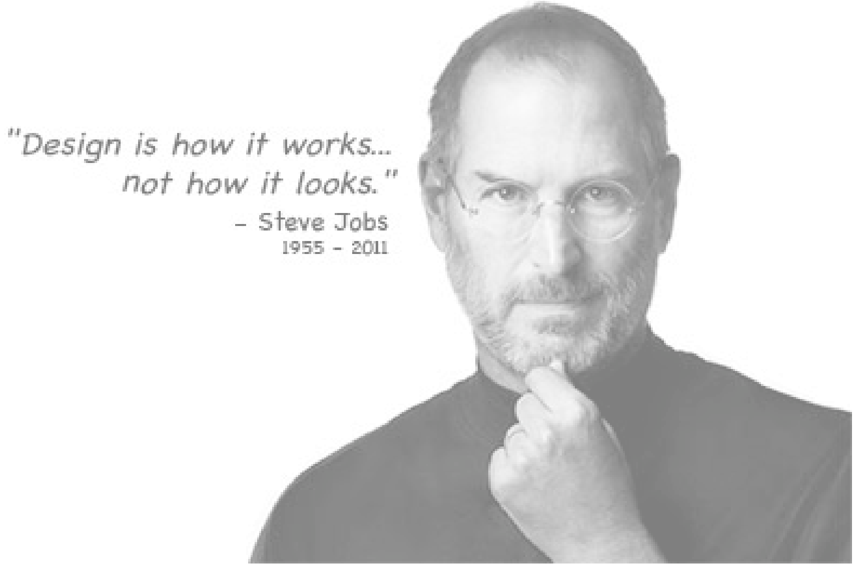
Software Design Facets

- software design facets :
  - also called views
  - representing a partial aspect of a software design that shows specific properties of a software system
  - pertain to distinct issues associated with software design
    - behavioural, functional, structural, data

- software design :
  - production of relatively independent and orthogonal multifaceted artefact

Software Design Concepts

- Abstraction
- Modularity
- Control Hierarchy
- Data Structure
- Information Hiding
- Refinement
- Software architecture
- Structural Partitioning
- Software Procedure

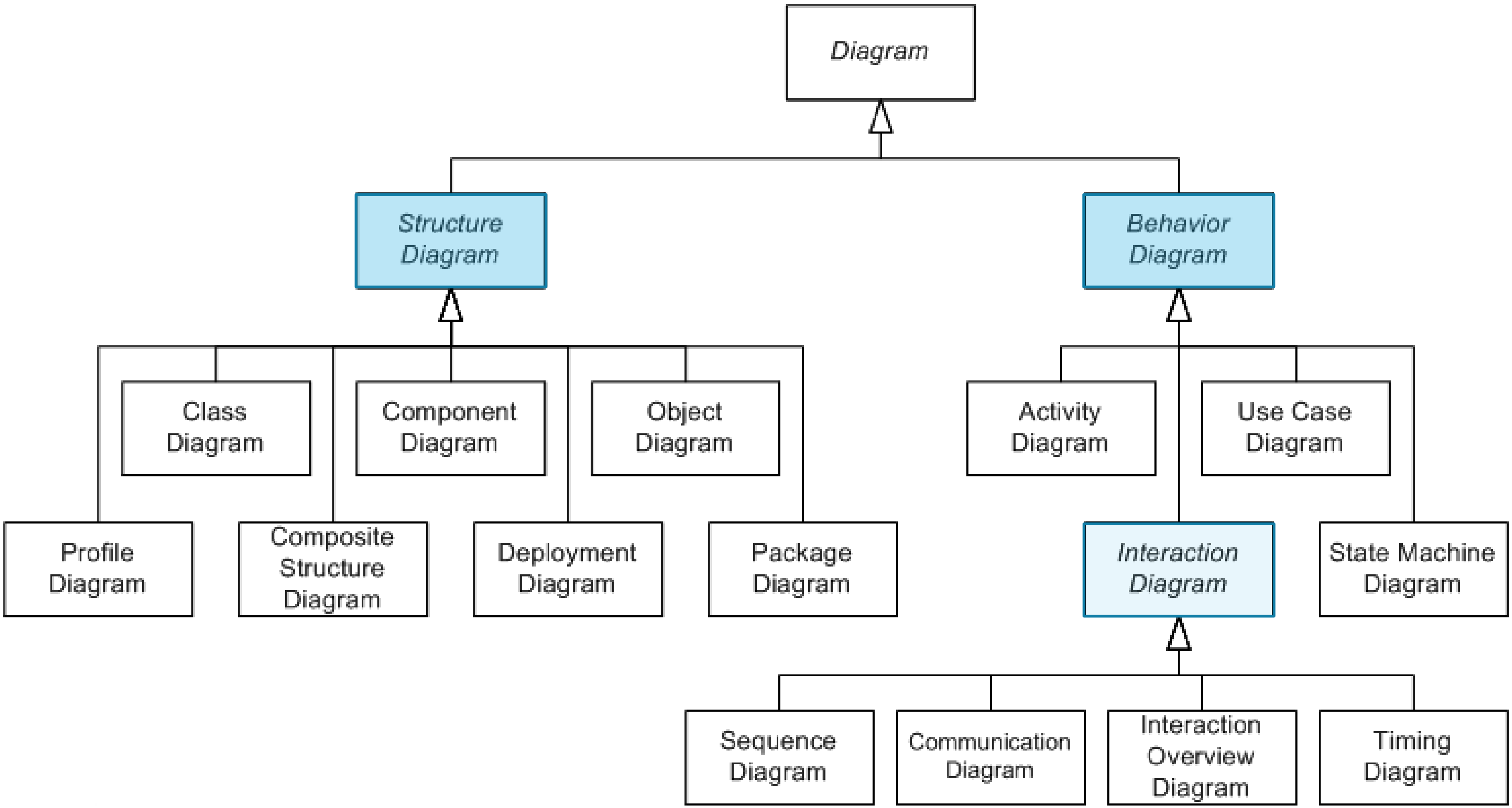


Software Design Considerations

- Compatibility
- Extensibility
- Modularity
- Fault-tolerance
- Maintainability
- Reusability
- Robustness
- Usability
- Performance
- Portability
- Scalability
- Reliability

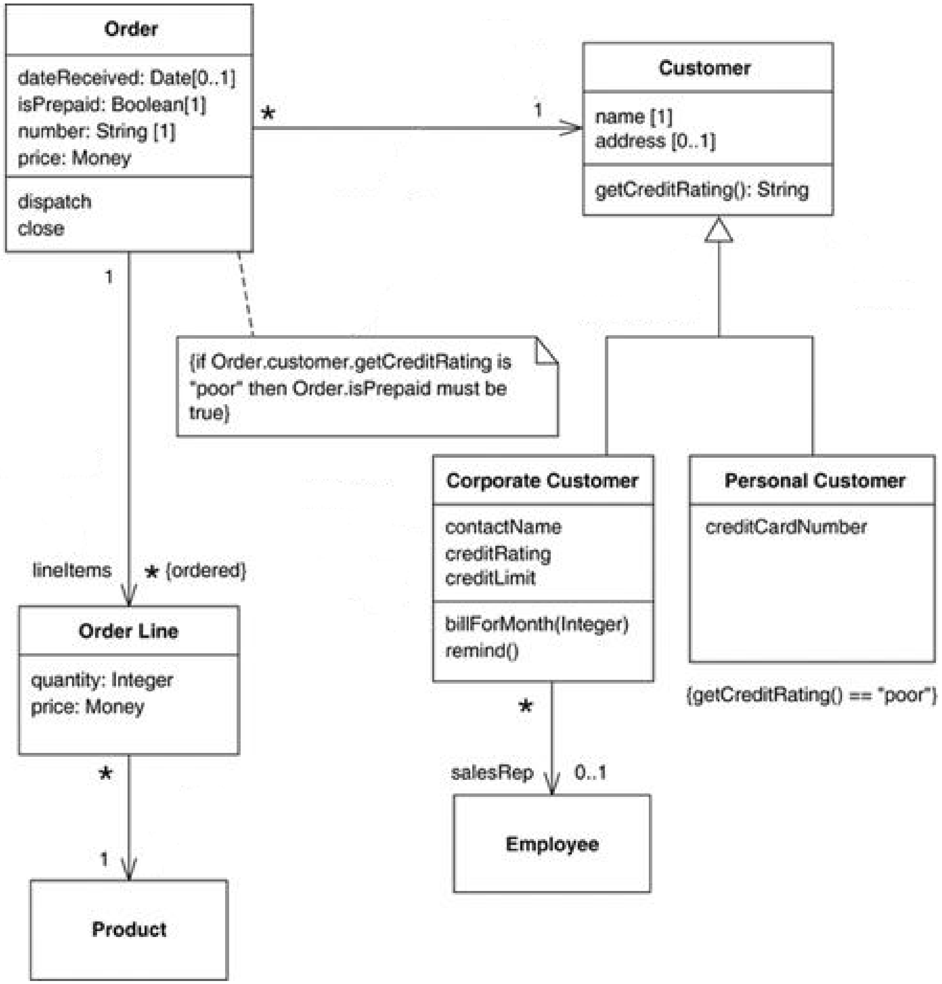
Software Design & UML

- Unified Modeling Language



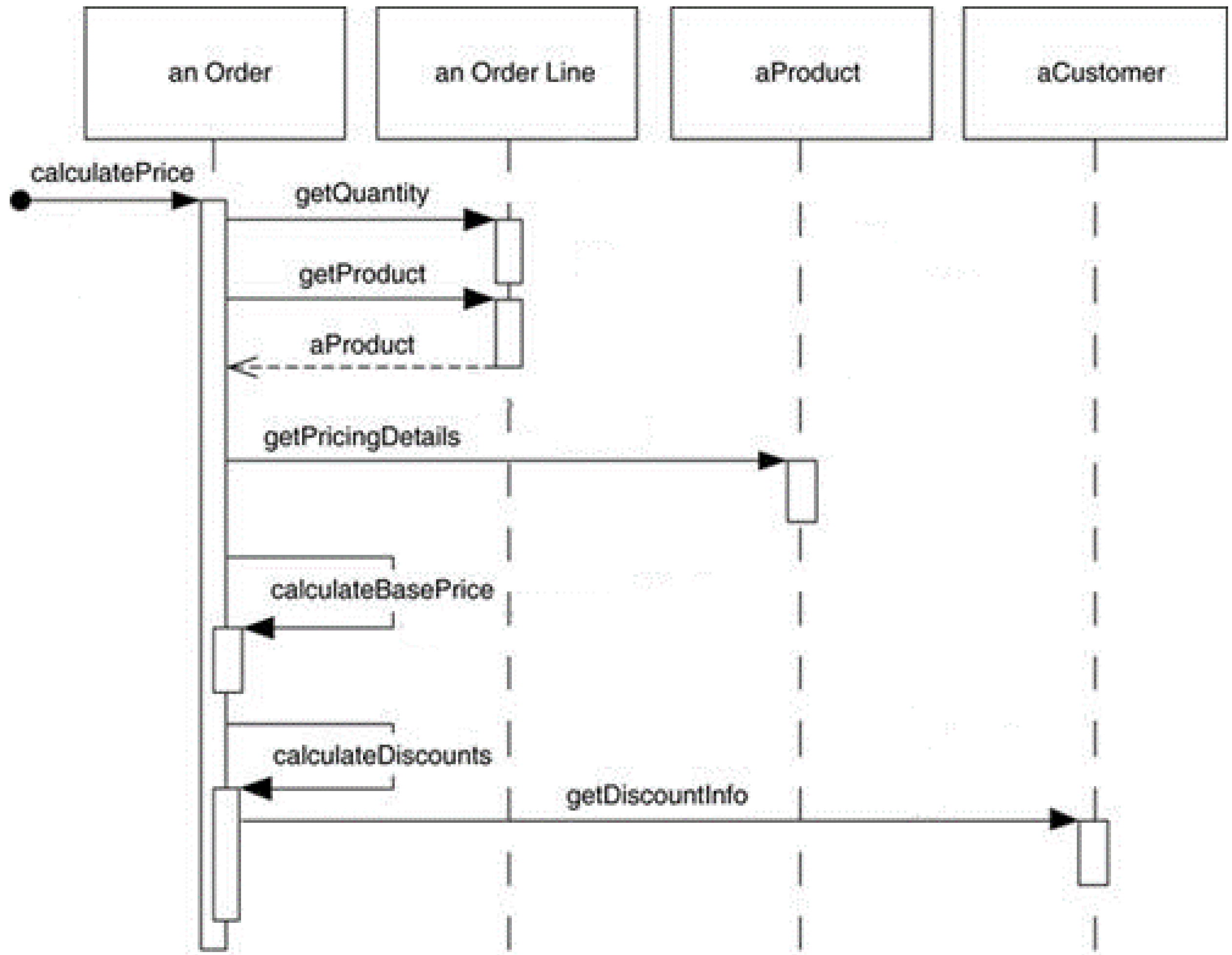
Software Design: Artefacts

- structural specification



Software Design: Artefacts

- behavioural specification

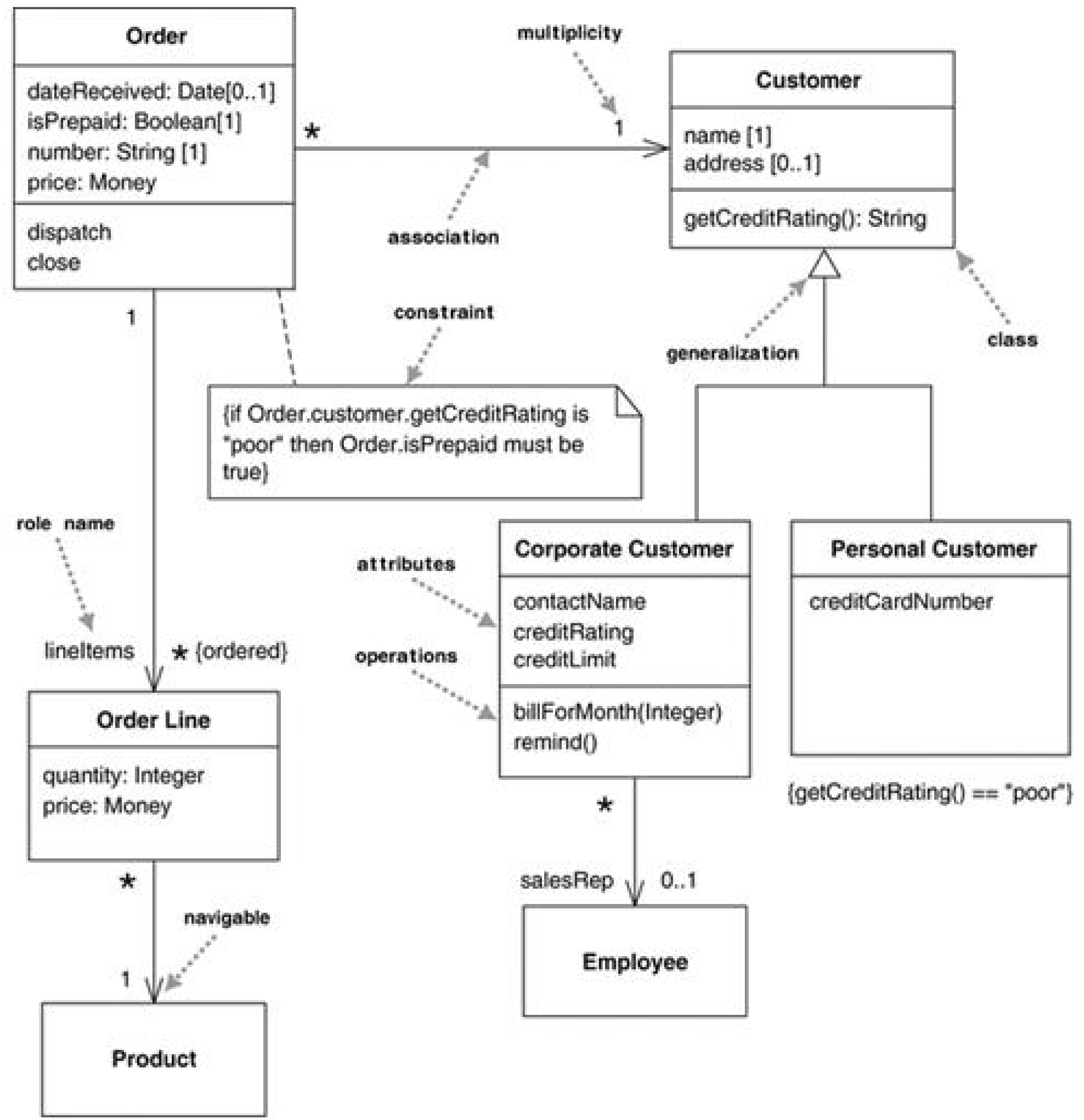


Design Recovery

STRUCTURAL

Class Diagram

Simple Class Diagram



Exercise : recover class diagram 1

- Employee.Java

```
public class Employee {

  private String name;
  private double payRate;
  private final int EMPLOYEE_ID;
  private static int nextID = 1000;
  public static final double STARTING_PAY_RATE = 7.75;

  public Employee(String name) {
    this.name = name;
    EMPLOYEE_ID = getNextID();
    payRate = STARTING_PAY_RATE;
  }

  public Employee(String name, double startingPay) {
    this.name = name;
    EMPLOYEE_ID = getNextID();
    payRate = startingPay;
  }
```

Exercise : recover class diagram 1

```
   public String getName() {
      return name;
   }
   public int getEmployeeID() {
      return EMPLOYEE_ID;
   }
   public double getPayRate() {
      return payRate;
   }
   public void changeName(String newName) {
      name = newName;
   }
   public void changePayRate(double newRate) {
      payRate = newRate;
   }
   public static int getNextID() {
      int id = nextID;
      nextID++;
      return id;
   }
}
```

Exercise : recover class diagram 1

Exercise : recover class diagram 2

- Driver.java

```
public class Driver {

  private StringContainer b = null;

  public static void main(String[] args){
    Driver d = new Driver();
    d.run();
  }

  public void run() {
    b = new StringContainer();
    b.add("One");
    b.add("Two");
    b.remove("One");
  }
}
```

- Vector.java (from java.util.Vector)

Exercise : recover class diagram 2

- StringContainer.java

```
import java.util.Vector;

public class StringContainer {
  private Vector v = null;

  public void add(String s) {
    init();
    v.add(s);
  }
  public boolean remove(String s) {
    init();
    return v.remove(s);
  }
  private void init() {
    if (v == null)
      v = new Vector();
  }
}
```

Exercise : recover class diagram 2

Exercise : recover class diagram 3

- Account.java, Bank.Java, BankSimulation.java

```
abstract class Account {
  protected int number;
  protected double bal;
  protected Person owner;
  public int getNumber() { … }
  public double getBal() { … }
  public Person getOwner() { … }
  public void deposit(double d) { … }
  public abstract boolean withdraw(double d);
}
```

```
class Bank {
  private Set<Account> accounts = new HashSet<Account>();
  public void addAccount(Account a) { … }
  public Account selectAccount(int no) { … }
}
```

```
class BankSimulation {
  public static void main(String[] args) { … }
}
```

Exercise : recover class diagram 3

- CheckingAccount.java, SavingsAccount.Java, Person.java

```
class CheckingAccount extends Account {
  private double chargeRate;
  public CheckingAccount(int no,double iR,Person o) { … }
  public boolean withdraw(double d) { … }
  public void payCharge() { … }
}
```

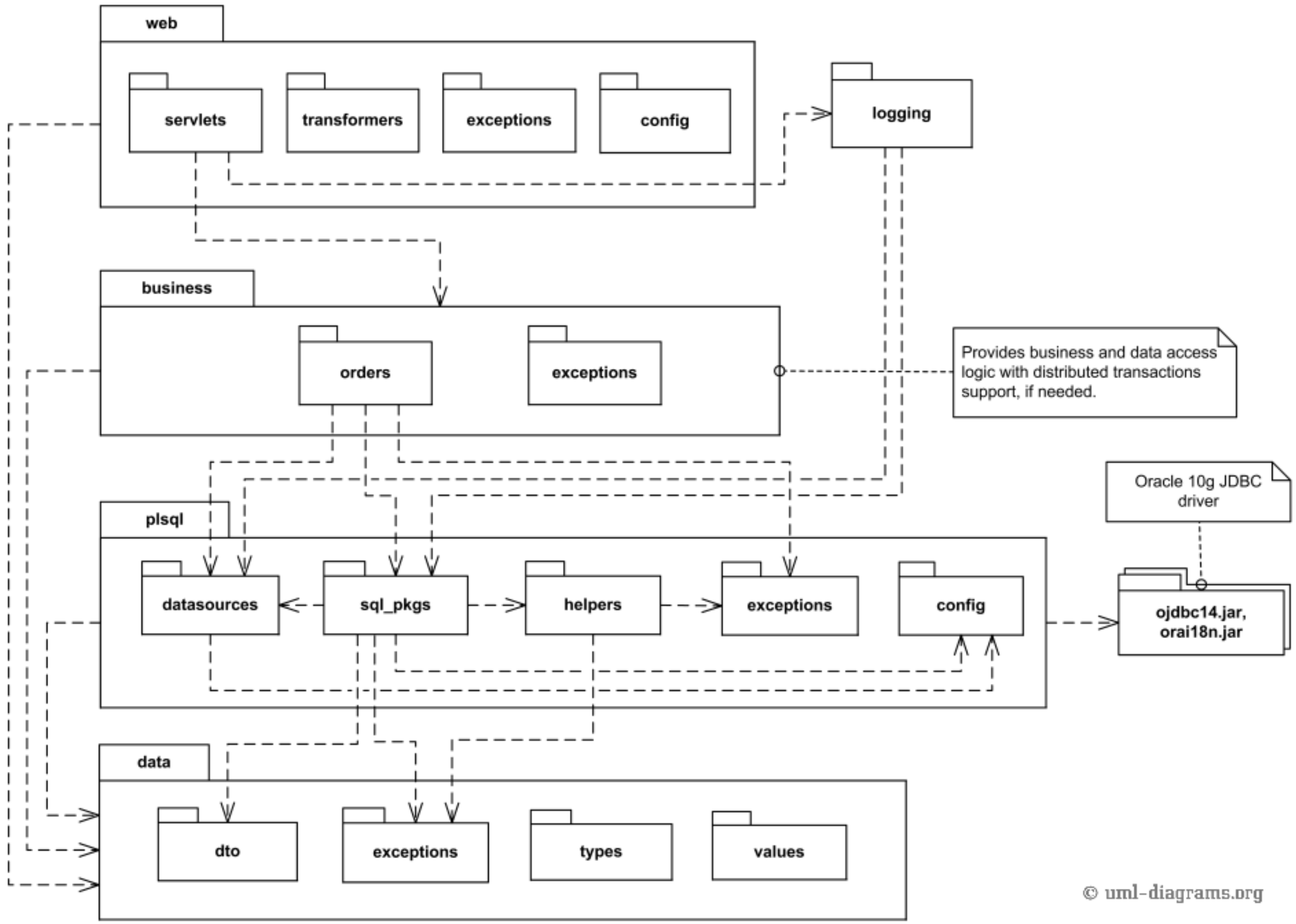
```
class SavingsAccount extends Account {
  private double interestRate;
  public SavingsAccount(int no,double iR,Person o) { … }
  public boolean withdraw(double d) { … }
  public void addInterest() { … }
}
```

```
class Person {
  private String name;
  private double salary;
  public Person(String n, double s) { … }
  public String getName() { … }
  public double getSalary() { … }
}
```

Exercise : recover class diagram 3

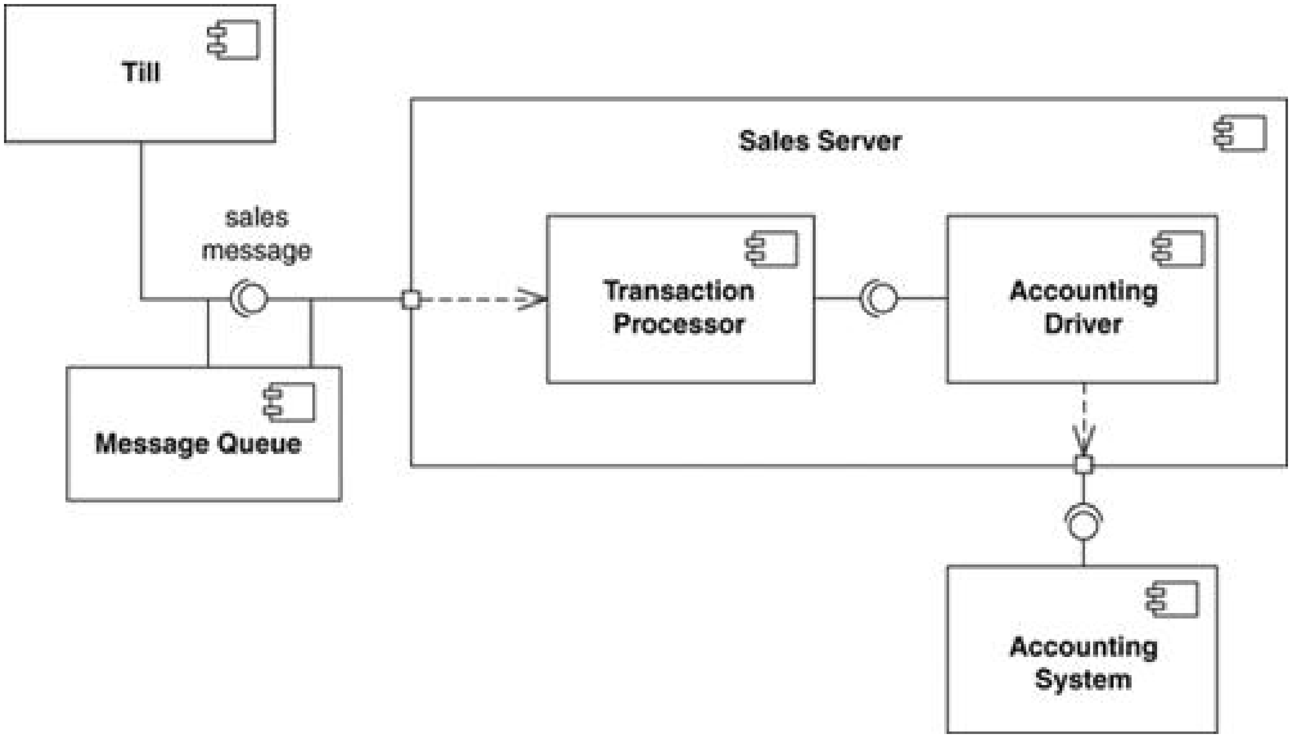
Package Diagram

Simple Package Diagram



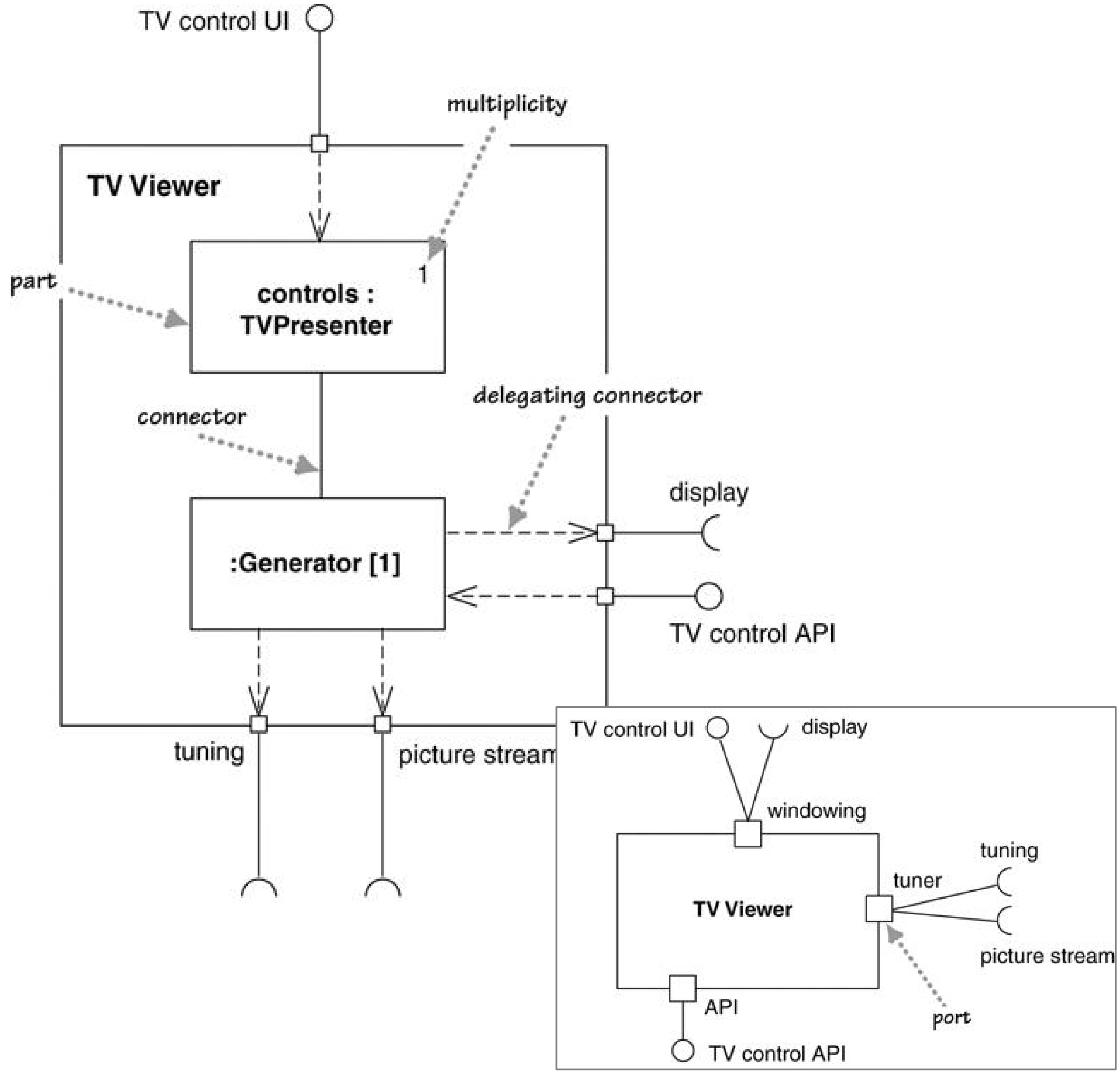
Component Diagram

Simple Component Diagram



Composite Structure Diagram

Simple Composite Structure

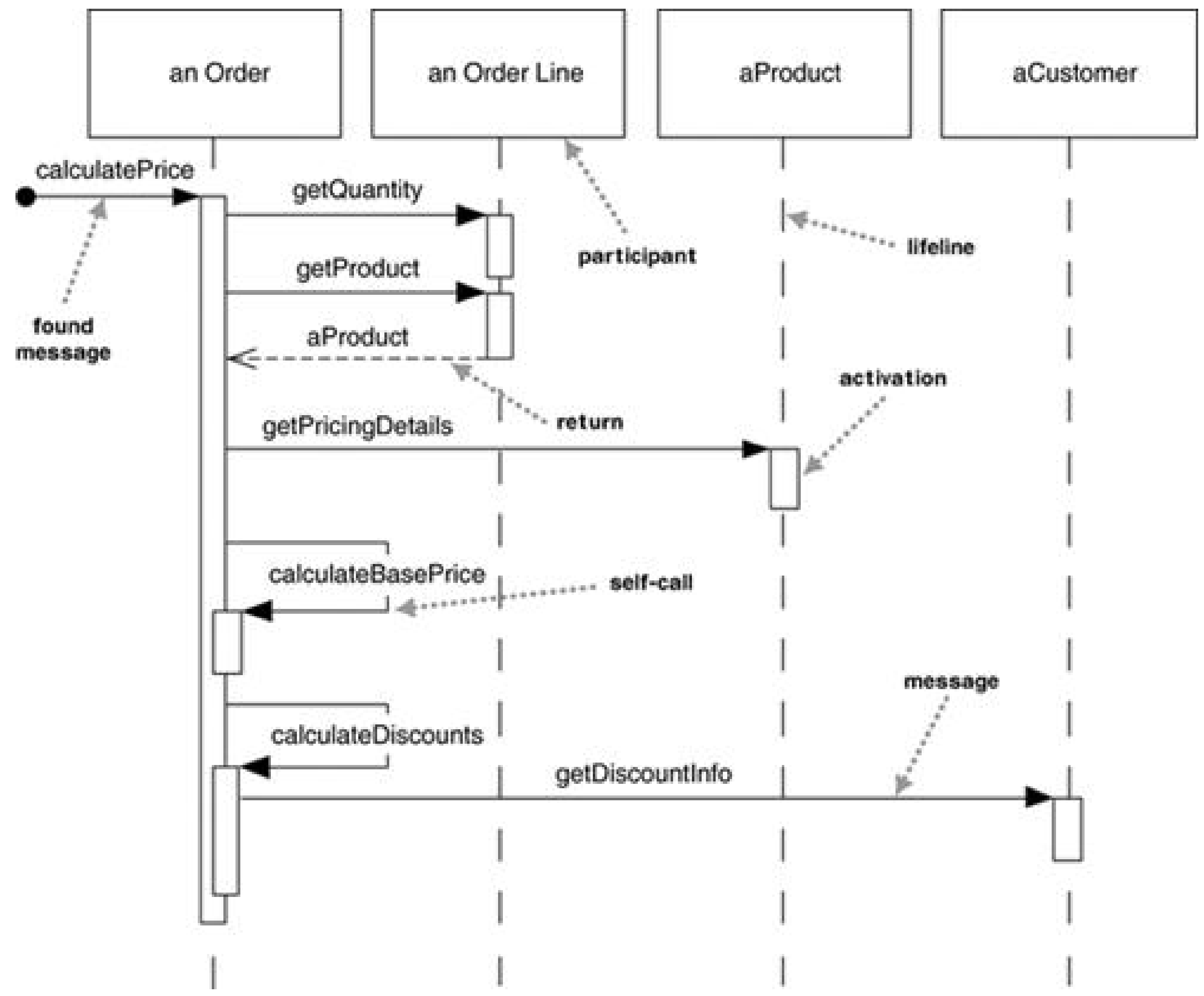


Design Recovery

BEHAVIOURAL

Sequence Diagram

Simple Sequence Diagram



Exercise : recover sequence diagram 1

- Client.java - work(), Device.Java, Server.java

```
public class Client {
  private Server server;
  public void work() {
    server.open();
    server.print("Hello");
    server.close();
  }
  …
```

```
class Device {
  public void write(String s) { … }
}
```

```
class Server {
  public Device device;
  public void open() { … }
  public void print(String s) {
    device.write(s);
    …
  }
  public void close() { … }
  …
```

Exercise : recover sequence diagram 1

Exercise : recover sequence diagram 2

- Driver.java – run()

```
public class Driver {

  private StringContainer b = null;

  public static void main(String[] args){
    Driver d = new Driver();
    d.run();
  }

  public void run() {
    b = new StringContainer();
    b.add("One");
    b.add("Two");
    b.remove("One");
  }
}
```

- Vector.java (from java.util.Vector)

Exercise : recover sequence diagram 2

- StringContainer.java

```
import java.util.Vector;

public class StringContainer {
  private Vector v = null;

  public void add(String s) {
    init();
    v.add(s);
  }
  public boolean remove(String s) {
    init();
    return v.remove(s);
  }
  private void init() {
    if (v == null)
      v = new Vector();
  }
}
```

Exercise : recover sequence diagram 2

Exercise : recover sequence diagram 3

- M.java – f(), Observer.java

```
public class M {

  public static void main(String[] args) {
    M m = new M();
    m.f();
  }

  public void f() {
    Subject s = new subject();
    Observer o1 = new Observer();
    Observer o2 = new Observer();
    s.addObserver(o1);
    s.addObserver(o1);
    s.changeState();
  }
}
```

```
class Observer {
  public void update() { }
}
```

Exercise : recover sequence diagram 3

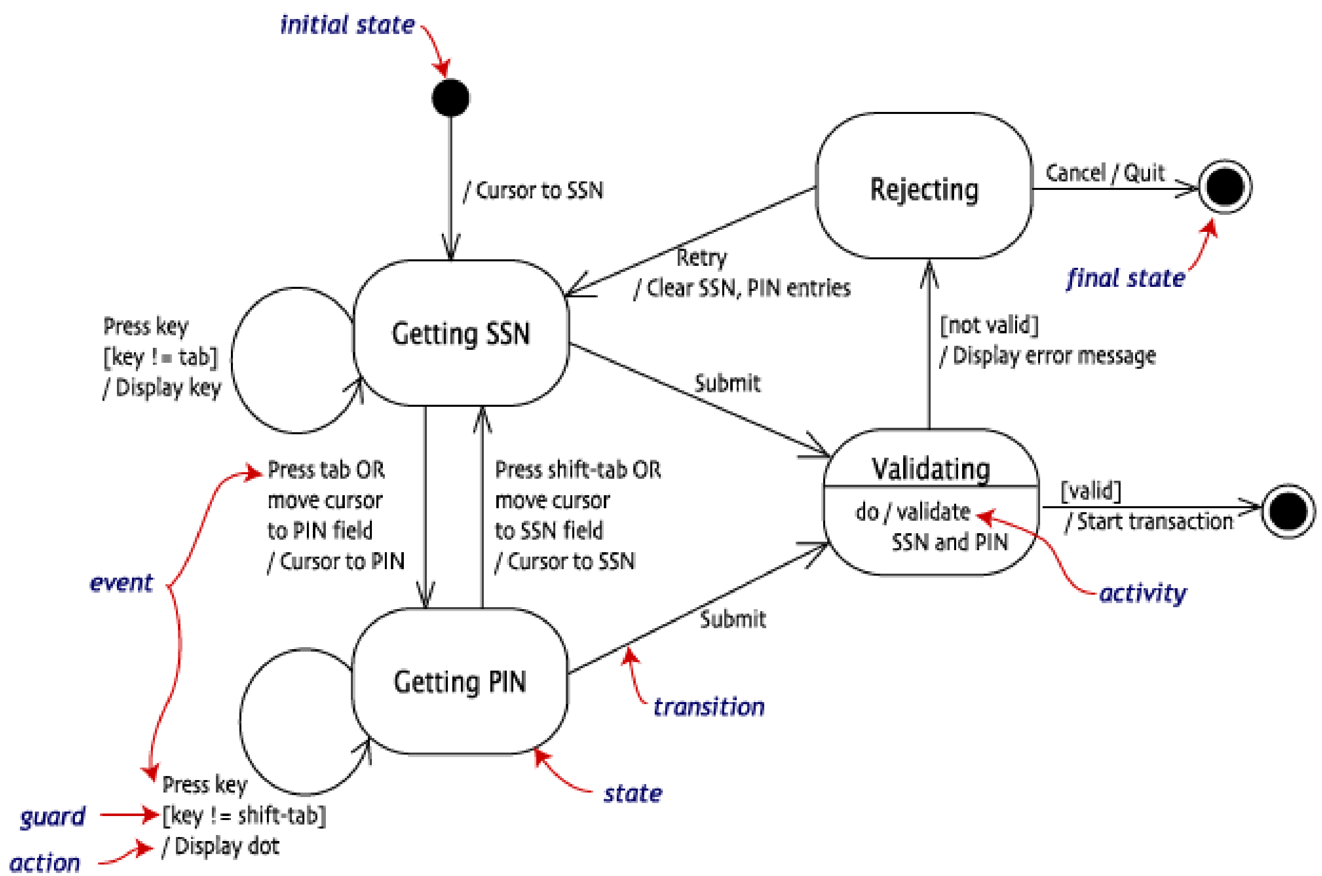
- Subject.java

```
public class Subject {
  private Collection c;

  public void addObserver(Observer o) {
    c.add(o);
  }

  public void changeState() {
    // change state of subject
    notifyObservers();
  }

  public void notifyObservers() {
    Iterator i = c.iterator();
      while (i.hasNext()) {
        Observer o = (Observer)i.next();
        o.update();
      }
  }
}
```

Exercise : recover sequence diagram 3

State Machine Diagram

Simple State Machine Diagram



Exercise : recover state machine diagram

```
public class SlidingDoor {
  private DoorState curState;
  public SlidingDoor() {
    curState = DoorState.INITIAL;
  }
  public void powerOn() {
    curState = DoorState.CLOSED;
  }
  public void objectDetected(int distance) {
    switch (curState) {
      case INITIAL :
        break;
      case CLOSED :
        if (distance <= 1) {
          door.open();
          curState = DoorState.BEING_OPENED ;
          startTimer(5);
        }
        break;
      case OPENED :
        startTimer(5);
        break ;
```

Exercise : recover state machine diagram

```
      case BEING_CLOSED:
        if (distance <= 1.2) {
          door.stopClosing();
          door.open();
          curState = DoorState.BEING_OPENED;
        }
        break;
      case BEING_OPENED:
        break;
    }
  }

  public void doorClosed() {
    switch (curState) {
      case INITIAL :
        break;
      case CLOSED :
        System.out.println("Unexpeted Event");
        curState = DoorState.FAILED;
        break ;
      case OPENED :
        break ;
```

Exercise : recover state machine diagram

```
      case BEING_CLOSED:
        door.stopClosing();
        curState = DoorState.CLOSED;
        break;
      case BEING_OPENED:
        break;
    }
  }

  public void doorOpened() {
    switch (curState) {
      case INITIAL :
        break;
      ...
      case BEING_OPENED:
        door.stopOpening();
        curState = DoorState.OPENED;
        startTimer(5);
        break;
    }
```

Exercise : recover state machine diagram

```
  public void timeout() {
    switch (curState) {
      case INITIAL :
        break;
      case CLOSED :
        break ;
      case OPENED :
        door.close();
        curState = DoorState.BEING_CLOSED;
        break ;
      ...
    }
  }
  ...
}

public enum DoorState {
  INITIAL, CLOSED, BEING_OPENED, OPENED, BEING_CLOSED, FAILED,
  FINAL
}
```

Exercise : recover state machine diagram

Architecture Recovery

Architecture Recovery

- also called
  - **architectural reconstruction**
  - **reverse architecting**

- definition :
  - unveil **design decisions** from system implementation and documentation
  - reverse engineering activities making **existing** of software **architectures explicit**
  - techniques and processes to **uncover** a system's **architecture**

Architecture Recovery - Purpose

- to **understand** software
  - identify design intent to modify legacy
  - understand cost and evaluate impact of change
  - staff turnover

- to support **redocumentation**

- to **re-engineer** / **renovate** architecture
  - design ideal architecture
  - discover (reverse) current architecture of legacy system
  - → rebalance to create architectural improvement plan

Architecture Recovery - Purpose

- to preserve **qualities**
  - intended vs. implemented architecture
  - software **evolution** & architectural **degradation**
    - architectural drift
      - increase brittleness / rigidity (resistance to change)
    - architectural erosion
      - need to enforce architecture

→ detect and resolve architectural problems

Architecture Recovery

- **software architecture**

*"… the **structure**(s) of the system, which comprises software **components**, the externally visible **properties** of those components, and the **relationships** among them."*

- Software Architecture in Practice

- to identify / understand architecture
  - architectural styles and views

Architecture Style

- 3 categories of styles
  - **module** :
    - introduce specific set of module types
    - specifies rules about how elements of those types can be combined
  - **C&C** :
    - specifies runtime behaviour in terms of components and connectors
    - specifies data and control flow
  - **allocation** :
    - specifies mapping of software units to elements of an development or execution environment

Architecture Style - Module

- decomposition style
  - show the structure of modules and submodules
- uses style
  - indicate functional dependency relations among modules
- generalization style
  - indicate specialization relations among modules
- layered style
  - describe the *allowed-to-use* relation in a restricted fashion between groups of modules called layers
- aspects style
  - describe particular modules called aspects that are responsible for crosscutting concerns
- data model style
  - used to show the relations among data entities

Architecture Style – C&C

- call-return styles
  - components interact through synchronous invocation of capabilities provided by other components
- data flow styles
  - computation is driven by the flow of data through the system
- event-based styles
  - components interact through asynchronous events or messages
- repository styles
  - components interact through large collections of persistent, shared data

Architecture Recovery Approach

1. **data gathering**
   - source code (static analysis)
   - historical information
   - human expertise
   - runtime behaviour (dynamic analysis)

2. **knowledge organization**
   - abstraction
     - aggregation & filtering to exclude useless information

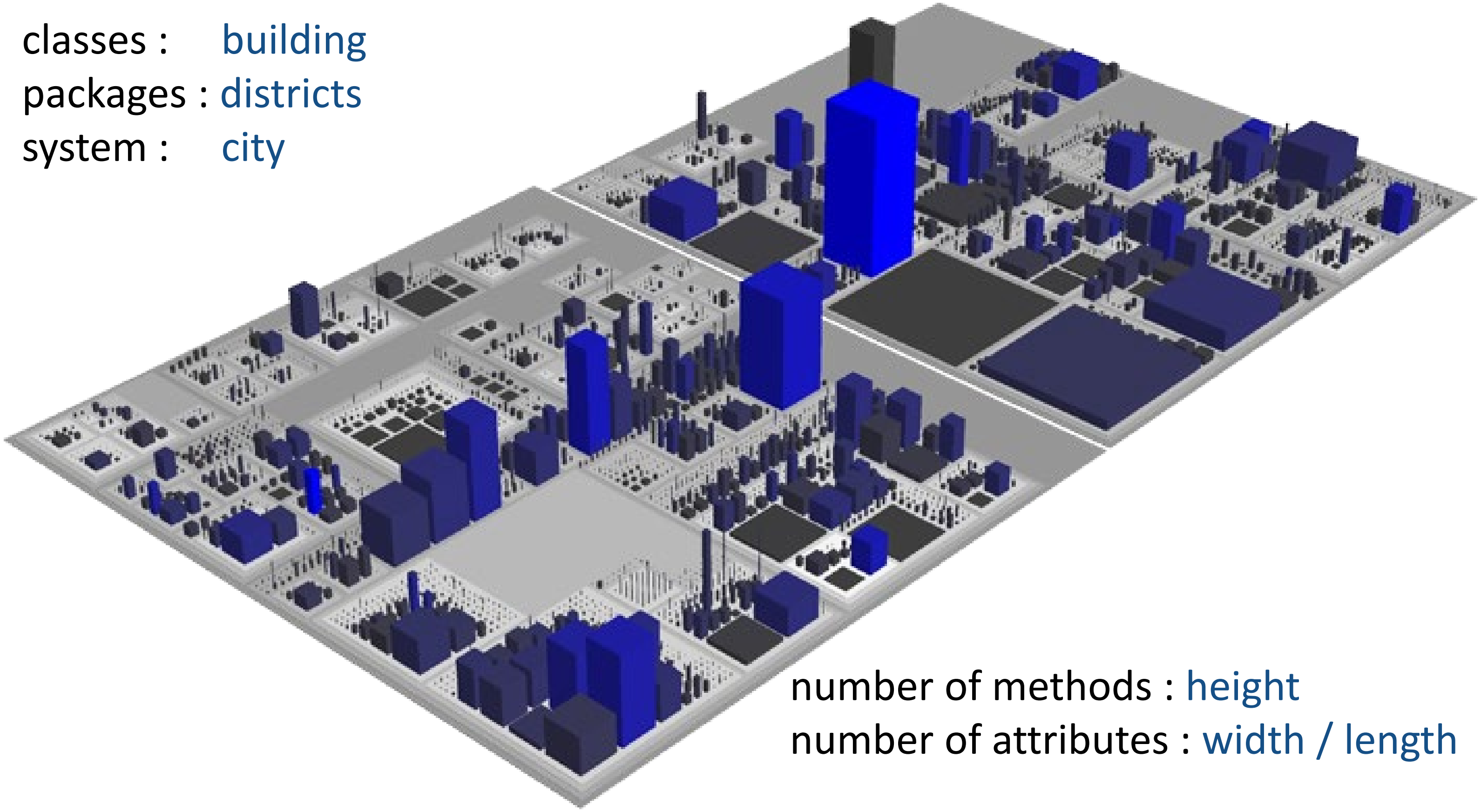
3. **information exploration**
   - navigation
   - analysing
   - presentation

Architecture Recovery Approach

- recover **module view** with **static dependency** checkers
  - Structure 101
  - Lattix
  - CppDepend
  - Ndepend
  - jDepend
  - ClassCycle
  - Dependency Finder
  - Understand
  - Bauhaus
  - SonarJ
  - Softwarenaut
  - STAN4J
  - Code City
- deduce real **structure**(s) with dependency analysis

Code City

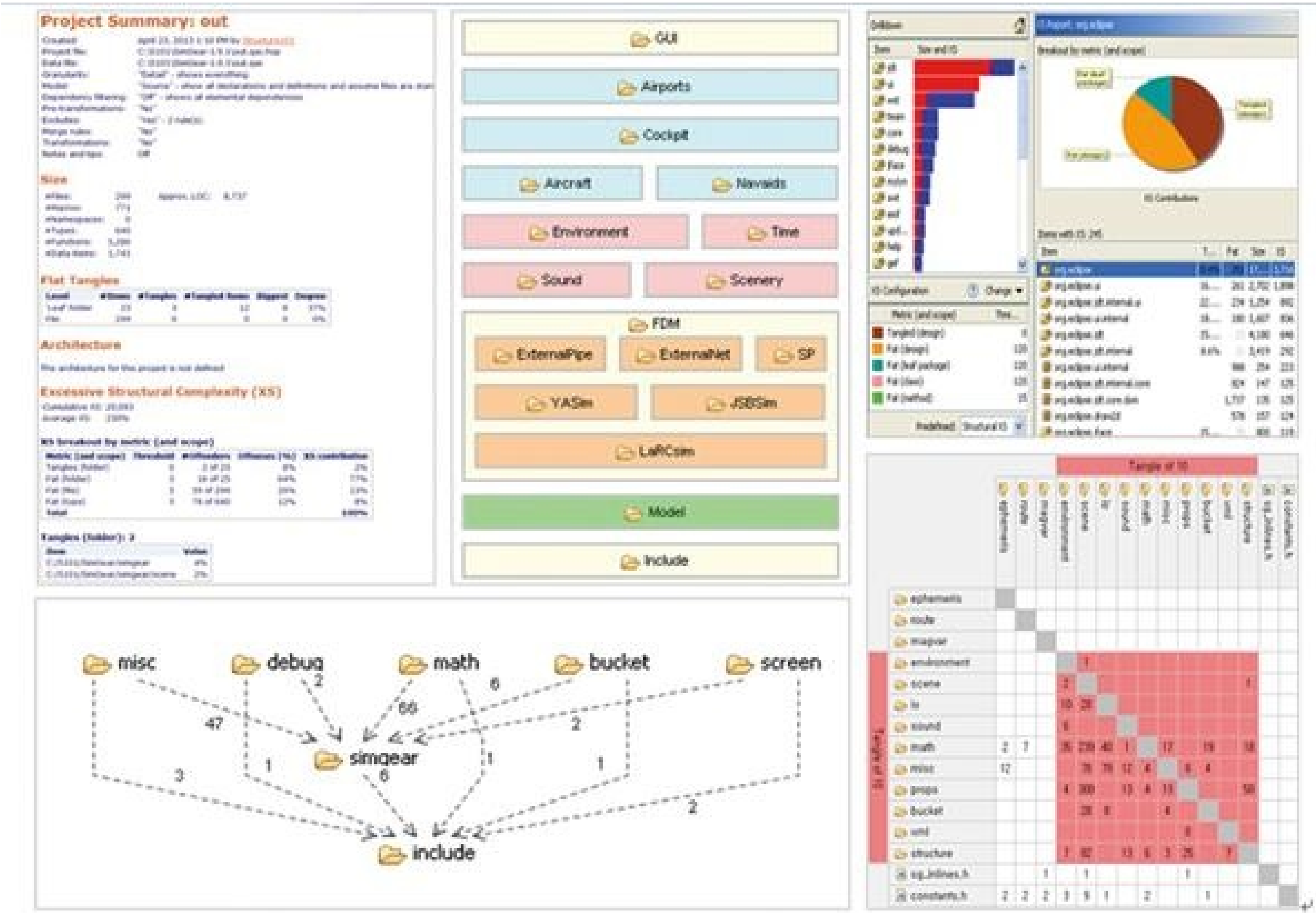
classes : building
packages : districts
system : city



number of methods : height
number of attributes : width / length

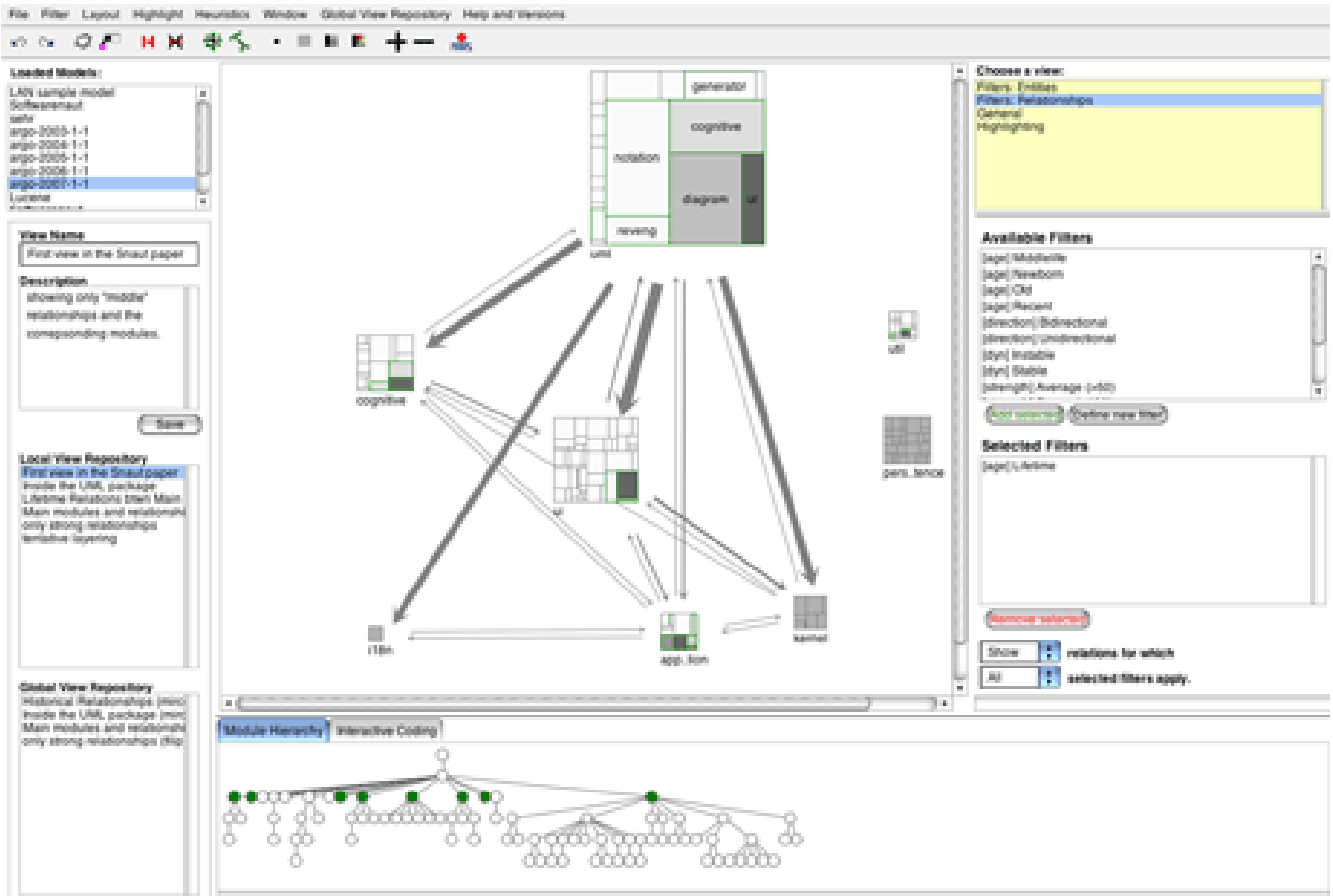
Visualization of JDK v1.5

Structure 101

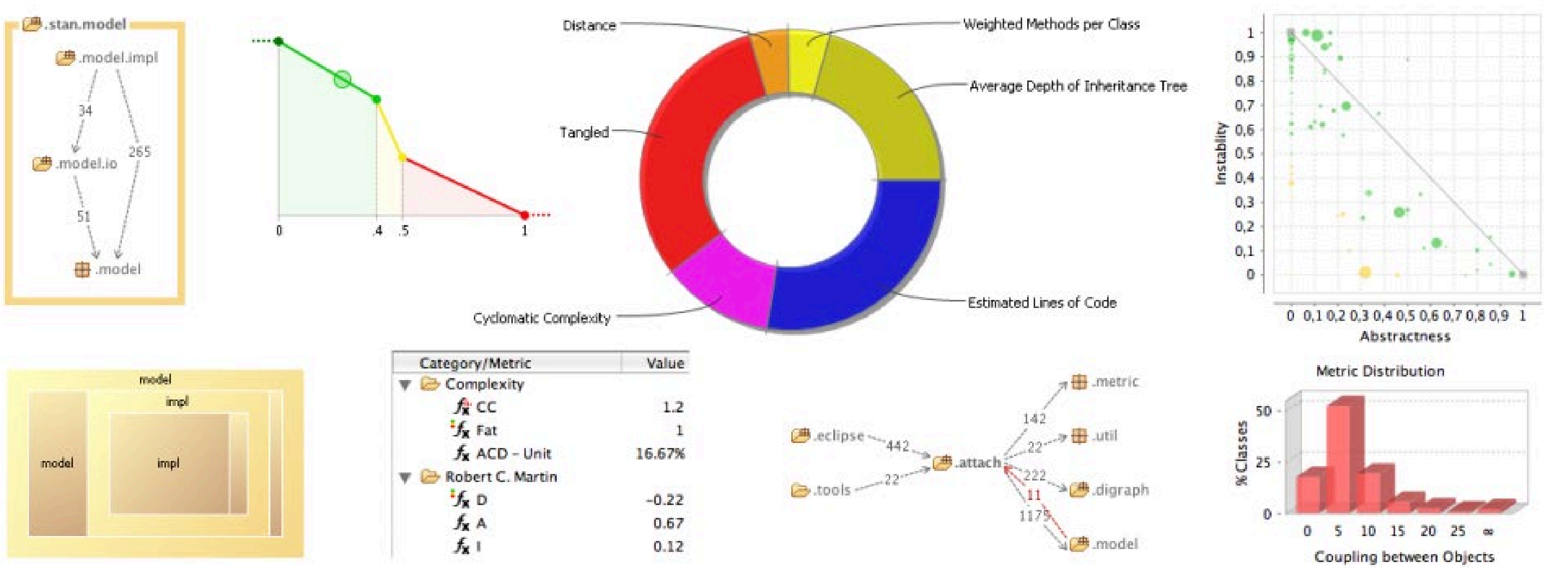


Understand Tool

Softwarenaut



STAN4J



.stan.model
.model.impl
34
265
.model.io
51
.model
0
.4
.5
1
Distance
Weighted Methods per Class
Average Depth of Inheritance Tree
Tangled
Cyclomatic Complexity
Estimated Lines of Code
Instability
Abstractness
model
impl
Category/Metric
Value
Complexity
CC 1.2
Fat 1
ACD - Unit 16.67%
Robert C. Martin
D -0.22
A 0.67
I 0.12
.eclipse
442
.tools
22
.attach
142
.metric
.util
222
11
1175
.digraph
Metric Distribution
% Classes
Coupling between Objects