Domain analysis

- Goal: build an object-oriented model of the real-world system (or imaginary world)
- Slicing the soup: OOA vs. OOD
  - OOA concerned with “what”, not “how”
  - OOA activities focus on the domain layer
- Common OOA activities: identify classes, assign (some) responsibilities to classes
  - Larman’s OOA: domain model (classes, associations, attributes), and system operations
    - Includes static and dynamic views of the domain
  - DA artifacts for CS 50 project: see assignment 3
Domain analysis activities

- **Static view** – model the domain
  - Identify domain concepts
  - Identify associations between the concepts
    - Now ready to start drawing domain model – a visual representation of these concepts and associations
  - Identify attributes of the concepts
    - Usually add to drawing (CS 50: add to class specifications)
- **Dynamic view** – model the system behavior
  - Make system sequence diagrams
  - Write system operation contracts
Identifying concepts

- **Class** = major abstraction (i.e., not just an attribute)
- **How to find candidate classes?**
  - Think/brainstorm about the domain
    - But *save the How?* questions for OOD
  - Use a **concept category list** — e.g., pp. 140-141 in text
  - Identify the nouns & noun phrases in problem statement, **use case** descriptions, other …
- **Consider all as candidates to start; refine later**
  - i.e., a candidate class turns out to be just an attribute
    - But common error to decide too early
Suggest: start CRC cards now

<table>
<thead>
<tr>
<th>Class (name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

- 1 card for each candidate class, showing:
  - Class name – do now
  - Responsibilities – knowledge now, operations in OOD
  - Collaborators – some now, more in OOD

- CRC cards are useful for both OOA and OOD:
  - OOA – help sort out classes; use to lay out diagrams
  - OOD – role-playing to find operations; more diagrams
Split cards into 3 piles

1. Critical classes – must include
2. Totally irrelevant classes – must reject
   - Set aside, but record as irrelevant in glossary
3. Classes you are still undecided about – ask yourself questions like the following:
   - Is it same as another class? Is it an instance?
   - Is it actually outside the system? (like a person)
   - Does it have unique knowledge/responsibilities?
   - Is it needed by other classes?
- Keep updating the piles as more is learned!
Choosing concept names

- Note: if you can’t think of a simple, clear name, maybe you have a bad abstraction!
- A good test: **Can a person with domain knowledge (not CS knowledge) describe the abstraction based on its name alone?**
- Best to use existing names “in the territory”
  - See Larman’s cartographer analogy (p. 145)
    - Also: “exclude irrelevant features” and “do not add things that are not there.”
- But no sense to labor over good candidate names
  - e.g., “register” vs. “POST” – Larman choice is arbitrary
Specification types

● Larman tip: types that specify attributes for other types are often handy (“Description Classes”)
  – e.g., a ProductDescription – includes UPC, price, and any other specs common to an Item

● Two main purposes:
  – Eliminate redundant storage – no need to store common specs with each Item
  – Prevents loss of info when objects depleted – i.e., when the last Item is sold

● In general, look for unifying concepts
Partial POS domain model

- a.k.a. static class diagram
- Concepts are boxes
- Associations are lines connecting boxes
- Other UML details to follow
Associations

- **Def:** relationships between concepts
- **Common associations:**
  - **Dependency** – a class “uses” another
  - **Generalization** – a class is derived from another
  - **Aggregation** – one class is a collection of others
  - But can be any kind of relationship
- **Good association names are important too**
  - And helpful to identify the direction of association
- **Also helpful to use proper UML**
UML: dependency relationship

- When a class “uses” or otherwise depends on another class to fulfill a responsibility
  - Dashed line with arrow in UML
UML: showing generalization

- a.k.a., inheritance – one class is derived from another
  - In UML, triangle at end of line “points” at parent class
• “Whole” is identified by the diamond shape at that end of the line
Naming associations

- Recommended for any relation between concepts
  - Absolutely necessary if UML lacks notation (like dependency, aggregation, or generalization)
- Use verb or verb phrase: e.g., “records”, “paid by”
Identifying associations

- Handy tool: common associations list – pp. 155-6
- Don’t overdo it
  - Useful associations only – otherwise clutter
  - Must be domain-meaningful at this stage
- Highest priority categories are “need-to-know” associations – knowledge of the relationship must be preserved for awhile
  - A is physically or logically part of B
  - A is physically or logically contained in B
  - A is recorded in B
Generalization

- A domain model term, concerning general-specific relationships
  - e.g., `Bird` – general – a.k.a. supertype
    `Penguin` – specific – a.k.a. subtype
    A Penguin *is a* Bird.

- Aids abstract thinking
- Facilitates handling
  - Express more economically in conceptual model
  - Lends itself to implementation using inheritance
    - Note: inheritance is a software term; not domain-related
When to use generalization

- Define a **subtype** of a concept when instances of the subtype differ from other instances, as in:
  - They have *additional* attributes, and/or associations
  - They are *handled differently*, in important ways
  - They represent things with *varying behaviors*

- Define a **supertype** to generalize concepts when:
  - All subtypes are *true* variations of a single concept,
  - Subtypes share the same attributes and associations,
  - And subtypes all conform to both:
    - 100% rule – all supertype attributes and associations apply
    - “is a” rule
Abstract Classes

- Def.: If every instance of a class $C$ must also be an instance of a subclass, then $C$ is called an abstract conceptual class.
If a Payment instance exists which is not a member of a subclass, then Payment is not abstract – it is **concrete**.

![Diagram showing relationships between Payment, CashPayment, CreditPayment, and CheckPayment classes](image-url)
UML: Abstract Classes

- UML notation: *italicized* class name

```
  Payment
  /  
 Cash Payment  Credit Payment  Check Payment
```
Class attributes

● a.k.a., “properties” of classes
  – Describe an object’s state at a point in time
  – Attributes are “pure data values” – not complex things (which are concepts, not attributes)

● Purpose of attribution:
  – Insure that all information needed by the system’s objects is remembered somewhere

● Encapsulation principles help guide attribution
  – Info is most useful if stored where it’s needed most
  – Identity info of an object is best stored with that object
More attribution principles

- What to store depends on the application
  - e.g. Employee – Name? Address? Wage? Title?
    - Key question: What does this application need?
  - i.e., need _pertinent abstractions_ of concepts

- Representation depends on application too
  - i.e., how to represent in the conceptual model
    - e.g., Title just a String? – okay – else if complex meaning, maybe it is a concept of its own, or an association

- Should be simple – “data types”
  - e.g., 5, “white” – has no unique identity
  - Note: an attribute may become implemented as a class
Attribute or Class?

- **Classes**: objects with unique identity
  - e.g., 2 instances of Person
- **Attributes**: primitive types
  - e.g., number, string, time…
- What to do with non-primitive data types?
  - composed of separate sections (address)
  - quantities with units (payment amount)
  - has more attributes (promotional price: start/end)
  - has operations associated (SSN: validate)
UML: Attribute or Class?

- Non-primitive data types may be shown as attributes or classes!

![UML Diagram]

```
<table>
<thead>
<tr>
<th>ProductSpecification</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ItemID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

or

```
<table>
<thead>
<tr>
<th>ProductSpecification</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: ItemID</td>
</tr>
</tbody>
</table>
```
Attribution in practice

- Two complementary approaches:
  1. Choose a class – list its properties
  2. Choose a property – ask what it describes
     - Do it both ways for a complete set of attributes
- Probably will discover new concepts
  - Okay – augment the conceptual model
  - Note: sometimes an association should store attributes
    - Means the association is a concept of its own
    - e.g., Gymnast, Team – and Membership to associate them
Attribution Pitfall

- Relate conceptual classes with an association, not an attribute!

![Diagram showing the relationship between Cashier and Register classes with attributes name and currentRegister, and number, connected by a 1:1 association.](image-url)
Glossary notes

- Record all attributes in the glossary
  - Sometimes called the “data dictionary”
- Also record all concepts, associations, operations, use cases, …
  - And any terms that require clarification
- Purpose: reduce risk of miscommunication
  - With clients, and other team members
  - And for yourself a few weeks down the road
  - And in CS 50 – so we can understand your artifacts
- But don’t overdo it – always minimize busywork
System behavior

- Focus is on dynamic view: states and sequences
- State of the system is like a snapshot – a point-in-time record of memory contents
  - What objects currently exist?
  - What associations are currently formed?
  - What are the current values of object attributes?
- System sequences involve changes in state
  - Objects are created and destroyed
  - Associations are formed and broken
  - Values of attributes are modified
System sequence diagrams

Partial SSD for Larman’s Buy Items use case

For all items, the Cashier records the UPC and quantity.

On completion of item entry, the Cashier indicates to the POST that the sale is complete.

The Cashier tells the Customer the total, and the Customer gives a payment to the Cashier.

The Cashier records the cash received amount.

Actor triggers each event

Note: System is an instance – and a “black box”

Each event has a signature
Naming events

- Use “level of intent” (still OOA, not OOD)
  - i.e., not committed to a particular design
    - e.g., makePayment instead of submitCash – leaves flexibility for other payment types (in later cycle)
- Start with a verb – signifies something to happen
- Be sure to cover each event in each use case
  - i.e., playGame() is not an event! – it is at least a whole use case; probably many events
  - Best place to look: use cases’ typical courses of events
- Tip: if a simple name doesn’t work – maybe trying to name a complex process, not an event
System operations

- Focus in analysis stage is on *effect* of operations
  - i.e., *what* happens to system’s state? – *not how*
- System operation contracts – describe the system’s response to events
  - Operation – same as event name; include parameters
  - Cross References – at least the use case(s) involved
  - Pre-conditions – *assumptions* about system state before the operation begins
  - Post-conditions – *end changes* the operation makes to system state: instances, attributes, associations
Contract Example

**Operation**: makePayment(amount: Money)

**Cross References**: UseCases: ProcessSale

**Preconditions**: A sale is underway.

**Postconditions**:
- a payment instance p was created
- p.amountTendered became amount
- p was associated with current Sale
- current Sale was associated with Store
Contract Guidelines

- Identify system operations from SSDs
- For complex operations (may have subtle results, unclear in use case): write contract
- For postconditions, use categories:
  - instance creation/deletion
  - attribute modification
  - associations formed & broken
- As usual: Don’t overdo it!