Transition: analysis to design

Now know "what"
Time to focus on "how"

Decided what to do
Might as well do it right
System design

● Goal, in general: solve the problem
  – Goal of OOD: convert OOA results into something that can be implemented
    ● e.g, as software (and/or hardware, services, …)

● Key considerations (a.k.a. tradeoffs):
  – Cost-effectiveness of solution vs. design/coding effort
    ● Can reduce effort by applying patterns, idioms, 3rd party, …
  – Reusability – maybe worth investing effort in
    ● Could save lots of effort later
    ● But can overly complicate a simple problem if overdone
Design in practice

- No "cookbook" method – no "right" way
  - But have some basic principles for guidance
  - And have a growing knowledge base on patterns

- Is an exercise in problem solving, so attack using the usual strategies
  - Divide/conquer – solve sub-problems to solve whole
  - Top-down approach, with stepwise refinements

- Unlike analysis – leave room for creativity
  - Concentration → incubation → inspiration
Design activities

- Consider "real" use cases
  - Sharpen focus to actual technology, specific user interfaces, particular other systems, ...
- Package coherent subsystems together
  - And organize the packages into overall system architecture
- Model the interactions between objects
  - Including interactions between packages
- See Draft Project: System Design
System architecture

- High-level descriptions of the system
  - Broad focus on significant structural elements
    - Subsystems, packages, interfaces to other systems
  - Level of detail all developers & stakeholders can follow
    - Use case views, deployment views, design views, …

- Many basic architecture types – vary by purpose
  - Pipes & Filters – for flexibility without user interaction
  - Repository – favor big data storage-retrieval systems
  - Layers – considered the "object-oriented architecture"
    - Much more to come …
Diagramming packages

- Groups of classes – good for architectural modeling
  - Abstraction benefit: lots of concepts modeled as one
    - A handy way to "divide and conquer" the problem

- Idea is to separate functional subsystems
  - Many associations among classes in same package
  - Few associations between packages

- Side benefit: team members can split work by packages
  - Works best with "clean" interfaces
Basic 3-tier architecture

- Can have many layers, but 3 are basic:
  1. **Presentation** layer – windows, reports, GUIs
  2. **Application logic** layer – domain, object services
  3. **Storage** layer – persistent data, basic services
Application logic layer partitions

- Partition by logical units (organize as packages)
  - Refer to collaborations on CRC cards – look for:
    - Minimal coupling between packages (few collaborations)
    - Highly cohesive within packages (many collaborations)
- CS 48: Agree on package interfaces as a team
  - Then split up the work accordingly
- At least split domain from service classes
  - e.g., report generators, database interfaces, offscreen graphics builders, …
About layered architectures

- Concept – each layer is a base for implementing layers above it
  - Ideally, knowledge and contact is one-way: down ↓
  - Lower layers don’t even know upper layers exist

- What are some good reasons to use layers?
  - Reduce complexity – separate the domain from the implementation as much as possible
  - Increase modifiability, and reuse potential
  - Easy to plug in off-the-shelf and 3rd party stuff
Layers can free up our thinking: e.g., wxWidgets architecture

<table>
<thead>
<tr>
<th>wxWidgets API</th>
<th>wxWidgets Port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>wxMSW</td>
</tr>
<tr>
<td></td>
<td>wxGTK</td>
</tr>
<tr>
<td></td>
<td>wxX11</td>
</tr>
<tr>
<td></td>
<td>wxMotif</td>
</tr>
<tr>
<td></td>
<td>wxMac</td>
</tr>
<tr>
<td></td>
<td>wxCocoa</td>
</tr>
<tr>
<td></td>
<td>wxOS2</td>
</tr>
<tr>
<td></td>
<td>wxPalmOS</td>
</tr>
<tr>
<td></td>
<td>wxMGL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Platform API</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win32</td>
<td>Windows/Windows CE</td>
</tr>
<tr>
<td>GTK+</td>
<td>Unix/Linux</td>
</tr>
<tr>
<td>Xlib</td>
<td>Mac OS 9/OS X</td>
</tr>
<tr>
<td>Motif/Lesstif</td>
<td>Mac OS X</td>
</tr>
<tr>
<td>Carbon</td>
<td>OS/2</td>
</tr>
<tr>
<td>Cocoa</td>
<td>Palm OS</td>
</tr>
<tr>
<td>PM</td>
<td>Unix/DOS</td>
</tr>
<tr>
<td>Palm OS Protein APIs</td>
<td></td>
</tr>
<tr>
<td>MGL</td>
<td></td>
</tr>
</tbody>
</table>
**Data services sub-layers**

- **Goal:** insulate domain classes from storage details
- **How?** – interface classes
- **Note:** often start design by choosing services (inc. software and hardware choices)

Diagram:

- Presentation
- Domain
  - Relational Database Interface
  - Communication
  - Object Database Interface
  - Reporting
- Application Frameworks & Support Libraries
- Examples:
  1. Java Applets, MFC Documents and Views, VisualAge Visual Parts
  2. JDK, MFC, STL
- Relational Database
- Object Database
Storage and network layer(s)

- The lowest and least coupled layers
- 3 main ways to store and/or transmit data
  - 1. Object databases, and remote object interaction
    - Most abstract, so easiest to adapt (high level access)
  - 2. Relational databases, and query-response sessions
    - Mid-level access (records ↔ objects) – need an interface
  - 3. Do-it-yourself schemes – lowest level access
- Best to decide early
  - And whether to buy or build new, adapt old, …
Separating models and views

- **Basic principle**: domain (model) never directly contacts the presentation (view)
  - But is ready to answer requests from the view
  - Or can contact indirectly by "broadcasting"

- **Related idea**: view should not control the domain
  - Okay if GUI signals an event (then model takes over)
  - Often use a mediator, an "application coordinator"
Model-view separation benefits

- Reuse model with different views
- Maybe reuse view with different models
- Have multiple views of the same model
  - Even simultaneously!
  - e.g., view model from many angles
- Side benefit – complexity management
  - Benefit here – don't have to worry about display while working on the model
- Reflects a recurring OOP themes – encapsulation and information hiding
About domain "controllers"

- Not usually a domain concept
  - Added to the model during design
- They tie the system to external events
  - e.g., classes a GUI will know about
- Common types:
  - Façade controller – represents whole system, overall business, "world" – e.g., an application coordinator
  - Role controller – mimics a real-world role
  - Use case controller – handles sequences of events, monitors use case progress
Interaction diagrams

- **Dynamic views** of interacting objects
  - Usually starts by system event (i.e., external message)
  - Receiving object either handles alone, or passes message along (internal messages)
- **Why bother diagramming interactions?**
  - Get big picture view – better design, code, system
  - Easier to change drawing than code
- Do together with class diagrams/specifications
  - Links in diagrams indicate visibility between classes
- **2 basic types:** sequence and communication
Sequence diagrams

- Use for simpler interactions – sequence easily shown as top-to-bottom interactions
Communication diagrams

- Handy for more complicated interactions – show sequences by numbering the interactions
Notation for interactions

- **Class vs. instance** –
  - *Sale* – class name for static methods only
  - `mySale:Sale` – object name:type for other

- **Messages** – shown along link line
  - Must number in communication diagram
  - Show parameters too (with optional types)
    - e.g., `2: cost:=price(amount:double)`
    - And return values if not void
      - e.g., `1.1: items:=count():int`

- **Iteration** – use * and optional [iteration clause]
  - e.g., `3*: [i:=1...10]li:=item(i):LineItem`
More notation for interactions

- Conditions – [condition:boolean]
  - e.g., \[1:[\text{new sale}]\text{create}() \rightarrow \text{POST}\text{-}\text{Sale}\]

- Use "stack" icon for multi-objects (collections)
  - Note: message may be to the collection object itself (e.g., a list), or to the individual elements if *

- Show algorithms as notes (dog-ear symbol)
  - But only need if tricky or otherwise relevant
  - Or if using a CASE tool that translates note to code
Design principles

- Not exactly "rules" – instead things to consider
  - Should lead to high quality designs
    - Easier to maintain, understand, reuse, and extend
    - e.g., expert, low coupling, high cohesion, do-it-myself
- Note: Larman labels some as "patterns"
  - General Responsibility Assignment Software Patterns
  - Says assigning responsibilities = "desert island skill"
    - Also notes: "one person's pattern is another's primitive building block" to acknowledge not exactly design patterns
The expert principle

- Assign responsibility to class that has the necessary information
  - i.e., the "information expert"
- Avoids passing info between objects
- Still have collaboration as objects help others
  - e.g., Sale knows about all LineItems, and LineItems know quantity (and get price from Specs)
    - So let LineItem calculate subtotal()
    - Sale accumulates total from subtotals
- Main benefit: encapsulation maintained
  - Easier to program, maintain, extend independently
Low coupling

- Minimize dependencies between classes
  - Note how expert principle does this too
  - e.g., Sale does not contact ProductSpecification directly
    - LineItem does that instead; otherwise, Sale needs parallel collection of ProductSpecifications

- So fundamental it influences all design decisions
  - Is an "evaluative" pattern – used to rate design quality

- Supports independent classes
  - More reusable, less subject to changes elsewhere, easier to program, …
High cohesion

- Refers to functional cohesion
  - Means no class does too much work – especially not a bunch of unrelated things
  - Basically should avoid "bloated" classes
    - Hard to understand, maintain, reuse, …
    - Usually means other classes should take some responsibilities
      - Like an overworked manager – should delegate more
- Rule of thumb: insure all parts of a class are somehow related – all attributes and operations
  - Working together to provide "well-bounded behavior"
- Benefits – the usual list, plus greater simplicity
Events, states, and transitions

- **Event** – a significant occurrence
  - e.g., telephone receiver taken “off hook”
- **State** – condition of an object at a moment in time (the time between events)
  - e.g., telephone “idle” between being placed on hook and taken off hook
- **Transition** – relationship between two states as an event occurs
  - e.g., when “off hook” event occurs, transition from “idle” to “active” state
State diagrams

- Purpose: to model the changing states of complex objects
Utility of state diagrams

- Normally not useful for internal events
  - An internal event is one that is triggered by an object inside the system boundary
  - Interaction diagrams already cover internal events
- Useful for monitoring (whole) system sequences
  - Idea is to model the changing system states during the course of a use case
- Previous students in the CS project class said they are very useful tools for making sure all important states and sequences are managed properly
  - So don’t just take the instructor’s word for it!
A use case statechart diagram

- Helps designer ensure things are done in the correct order
More GRASP principles

- **Polymorphism** – if behavior varies *by type*
  - Assign responsibility for the variation to the types
    - Do not test for type or use other conditional logic!

- **Indirection** – to reduce coupling
  - Assign responsibility to *intermediate* class or interface

- **Pure fabrication** – artificial, non-domain class
  - e.g., encapsulate a cohesive set of responsibilities

- **Protected variations** – for variable/unstable parts
  - Assign responsibilities to *stable* interfaces
Software realities

- **Do-it-myself principle** (a.k.a., animation pattern)
  - Objects must do for themselves what normally is done to the real world objects they represent
  - e.g., in real world, somebody draws the figure – in software, figure draws itself: `figure.draw()`
  - Another e.g., `trajectory.map()` – normally would be mapped by outside observer if at all

- **Assume/insure basic services** are always available
  - i.e., get/set for attributes, add/remove/… for lists, …
  - So no need to include in class diagrams or specs
Inheritance – a software idea

- An object-oriented software construct for implementing generalization relations
  - Can reuse code by inheriting it with new code
- Allows consistent handling of different subtypes
  - As long as they have a common supertype
- But can be overdone!
  - Common error: forcing an “is a” relationship
    - e.g., class Easel : public Canvas – okay, but limited, because Easel cannot inherit from any other class now
  - Alternative is composition
    - e.g., more flexible to let Easel have a Canvas to draw on
Abstract types

- Always supertypes, by definition
  - Have no concrete existence in model
  - Definition – class A is an abstract type if every instance of A must be a subtype of A
  - e.g., Thing – an abstract type
    - How to draw a Thing? Describe a Thing? …
    - Must have a concrete Thing to draw, describe, …
  - Certain operations must be implemented by subtypes

- Abstract types are central to many design patterns
  - pure abstractions are more flexible than concrete types
  - actually just define interfaces for “families” of types
A note about subtypes & states

- Avoid using subtypes of a concept to represent changing states of that concept
  - Usually better to consider a State concept
    - State is an abstract type – with concrete subtypes
    - The original concept “is in” one State or another

- Exception is when it really makes sense to do
  - e.g., a Caterpillar becomes a Butterfly
  - i.e., a complete metamorphosis – change in state results in different attributes and associations
Design patterns summary

- “Tricks of the trade” for OO designers
  - Tried and true solutions to recurrent problems
    - Generally apply to various situations – e.g., Façade Pattern
  - Usually reflect basic design principles
- “Gang of Four” (GoF) patterns – seminal catalog
  - Four essential elements:
    1. A meaningful name – elevates thought to higher abstraction
    2. A problem description – where the pattern can apply
    3. The solution – like a template to apply the pattern
    4. Consequences – results and trade-offs
- Recurring theme: “encapsulate what varies most”
Types of GoF design patterns

- 7 are *structural* patterns – composition of classes/objects
  - e.g., Adapter
    - Problem: tool has interface X, client prefers interface Y
    - Solution: Adapter satisfies X, but looks like Y
    - Consequences: don’t reprogram X, and don’t distort Y to satisfy X
  - Bridge, Composite, Decorator, Façade, Flyweight and Proxy
- 5 are *creational* patterns – for creating objects
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton
- 11 are *behavioral* patterns – ways classes/objects interact
  - e.g., Chain of Responsibility, Command, and … 9 more
Creational design patterns

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

- **Abstract Factory**
  Creates an instance of several families of classes

- **Builder**
  Separates object construction from its representation

- **Factory Method**
  Creates an instance of several derived classes

- **Object Pool**
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

- **Prototype**
  A fully initialized instance to be copied or cloned

- **Singleton**
  A class of which only a single instance can exist
**Structural design patterns**

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

- **Adapter**
  Match interfaces of different classes

- **Bridge**
  Separates an object’s interface from its implementation

- **Composite**
  A tree structure of simple and composite objects

- **Decorator**
  Add responsibilities to objects dynamically

- **Facade**
  A single class that represents an entire subsystem

- **Flyweight**
  A fine-grained instance used for efficient sharing

- **Private Class Data**
  Restricts accessor/mutator access

- **Proxy**
  An object representing another object
Behavioral design patterns

These design patterns are all about Class’s objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

- **Chain of responsibility**
  A way of passing a request between a chain of objects

- **Command**
  Encapsulate a command request as an object

- **Interpreter**
  A way to include language elements in a program

- **Iterator**
  Sequentially access the elements of a collection

- **Mediator**
  Defines simplified communication between classes

- **Memento**
  Capture and restore an object’s internal state

- **Null Object**
  Designed to act as a default value of an object

- **Observer**
  A way of notifying change to a number of classes

- **State**
  Alter an object’s behavior when its state changes

- **Strategy**
  Encapsulates an algorithm inside a class

- **Template method**
  Defer the exact steps of an algorithm to a subclass

- **Visitor**
  Defines a new operation to a class without change
User interface design

- Major goal: match the skills, experience and expectations of its anticipated users
- Consider “human factors”
  - People have limited short-term memory, they make mistakes, and they are not all the same
- Some basic principles of UI design
  - User-oriented, not computer-oriented
  - Consistency – and especially minimal surprise
  - Recoverability, and guidance
User Interface issues

- Two fundamental problems to solve
  - How should information from the user be provided to the computer system?
  - How should information from the computer system be presented to the user?
- Many interaction styles – each has a place
  - Direct manipulation
  - Menu selection
  - Form fill-in
  - Commands – and (ideally) natural language
Sometimes multiple interfaces

Graphical user interface (Gnome/KDE)

X-windows GUI manager

Linux operating system

Unix shell interface (ksh/csh)

Command language interpreter

Figure from Ian Sommerville, *Software Engineering 8th edition*, Chapter 16
UI design process

Figure from Ian Sommerville, *Software Engineering 8th edition*, Chapter 16