The Software Design Process

CPSC 315 – Programming Studio

Outline

- Challenges in Design
- Design Concepts
- Heuristics
- Practices

Challenges in Design

- A problem that can only be defined by solving it
  - Only after “solving” it do you understand what the needs actually are
  - e.g. Tacoma Narrows bridge design
  - “Plan to throw one away”

- Process is Sloppy
  - Mistakes
  - Wrong, dead-end paths
  - Stop when “good enough”

- Tradeoffs and Priorities
  - Determine whether design is good
  - Priorities can change
### Challenges in Design

- **Restrictions are necessary**
  - Constraints improve the result
- **Nondeterministic process**
  - Not one “right” solution
- **A Heuristic process**
  - Rules of thumb vs. fixed process
- **Emergent**
  - Evolve and improve during design, coding

### Levels of Design

- **Software system as a whole**
- **Division into subsystems/packages**
- **Classes within packages**
- **Data and routines within classes**
- **Internal routine design**
  - Work at one level can affect those below and above.
  - Design can be iterated at each level

### Key Design Concepts

- **Most Important: Manage Complexity**
  - Software already involves conceptual hierarchies, abstraction
  - Goal: minimize how much of a program you have to think about at once
  - Should completely understand the impact of code changes in one area on other areas

### Good Design Characteristics

- **Minimal complexity**
- **Favor “simple” over “clever”**
Good Design Characteristics

- Minimal complexity
- Ease of maintenance

- Imagine what maintainer of code will want to know
- Be self-explanatory

- Keep connections between parts of programs minimized
  - Avoid n² interactions!
  - Abstraction, encapsulation, information hiding

Good Design Characteristics

- Minimal complexity
- Ease of maintenance
- Loose coupling

- Should be able to add to one part of system without affecting others

- Design so code could be “lifted” into a different system
- Good design, even if never reused

Good Design Characteristics

- Minimal complexity
- Ease of maintenance
- Loose coupling
- Extensibility

- Should be able to add to one part of system without affecting others

- Design so code could be “lifted” into a different system
- Good design, even if never reused

Good Design Characteristics

- Minimal complexity
- Ease of maintenance
- Loose coupling
- Extensibility
- Reusability

- Should be able to add to one part of system without affecting others

- Design so code could be “lifted” into a different system
- Good design, even if never reused
<table>
<thead>
<tr>
<th>Good Design Characteristics</th>
<th>Good Design Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Minimal complexity</td>
<td>• Minimal complexity</td>
</tr>
<tr>
<td>• Ease of maintenance</td>
<td>• Ease of maintenance</td>
</tr>
<tr>
<td>• Loose coupling</td>
<td>• Loose coupling</td>
</tr>
<tr>
<td>• Extensibility</td>
<td>• Extensibility</td>
</tr>
<tr>
<td>• Reusability</td>
<td>• Reusability</td>
</tr>
<tr>
<td>• High fan-in</td>
<td>• High fan-in</td>
</tr>
<tr>
<td>• Low-to-medium fan-out</td>
<td>• Low-to-medium fan-out</td>
</tr>
<tr>
<td>• Portability</td>
<td>• Complexity management</td>
</tr>
</tbody>
</table>

For a given class, have it used by many others. Indicates good capture of underlying functions.

Don’t use too many other classes.

Consider what will happen if moved to another environment.

Don’t add extra parts. Extra code will need to be tested, reviewed in future changes.

Leanness

Don’t add extra parts.
Good Design Characteristics

- Minimal complexity
- Ease of maintenance
- Loose coupling
- Extensibility
- Reusability
- High fan-in
- Low-to-medium fan-out
- Portability
- Leanness
- Stratification

Design so that you don't have to consider beyond the current layer

Stratification

Standard Techniques

- Use of common approaches make it easier to follow code later
- Avoid unneeded exotic approaches

Find Real-World Objects

- Standard Object-Oriented approach
- Identify objects and their attributes
- Determine what can be done to each object
- Determine what each object is allowed to do to other objects
- Determine the parts of each object that will be visible to other objects (public/private)
- Define each object's public interface

Design Heuristics

- Rules-of-thumb
  - “Trials in Trial-and-Error”
- Understand the Problem
- Devise a Plan
- Carry Out the Plan
- Look Back and Iterate
Form Consistent Abstractions

- View concepts in the aggregate
  - “Car” rather than “engine, body, wheels, etc.”
- Identify common attributes
  - Form base class
- Focus on interface rather than implementation
- Form abstractions at all levels
  - Car, Engine, Piston

Inheritance

- Inherit *when helpful*
  - When there are common features

Information Hiding

- Interface should reveal little about inner workings
  - Example: Assign ID numbers
    - Assignment algorithm could be hidden
    - ID number could be typed
  - Encapsulate Implementation Details
- Don’t set interface based on what’s easiest to use
  - Tends to expose too much of interior
- Think about “What needs to be hidden”

More on Information Hiding

- Two main advantages
  - Easier to comprehend complexity
  - Localized effects allow local changes
- Issues:
  - Circular dependencies
    - A->B->A
  - Global data (or too-large classes)
  - Performance penalties
    - Valid, but less important, at least at first
Identify Areas Likely to Change

- Anticipate Change
  - Identify items that seem likely to change
  - Separate these items into their own class
  - Limit connections to that class, or create interface that’s unlikely to change
- Examples of main potential problems:
  Business Rules, Hardware Dependencies, Input/Output, Nonstandard language features, status variables, difficult design/coding areas

Keep Coupling Loose

- Relations to other classes/routines
- Small Size
  - Fewer parameters, methods
- Visible
  - Avoid interactions via global variables
- Flexible
  - Don’t add unnecessary dependencies
  - e.g. using method that’s not unique to the class it belongs to

Kinds of Coupling

- Data-parameter (good)
  - Data passed through parameter lists
  - Primitive data types
- Simple-object (good)
  - Module instantiates that object
- Object-parameter (so-so)
  - Object 1 requires Object 2 to pass an Object 3
- Semantic (bad)
  - One object makes use of semantic information about the inner workings of another

Examples of Semantic Coupling

- Module 1 passes control flag to Module 2
  - Can be OK if control flag is typed
- Module 2 uses global data that Module 1 modifies
- Module 2 relies on knowledge that Module 1 calls initialize internally, so it doesn’t call it
- Module 1 passes Object to Module 2, but only initializes the parts of Object it knows Module 2 needs
- Module 1 passes a Base Object, but Module 2 knows it is actually a Derived Object, so it typecasts and calls methods unique to the derived object
Design Patterns

- *Design Patterns*, by “Gang of Four” (Gamma, Helm, Johnson, Vlissides)
- Common software problems and solutions that fall into patterns
- Provide ready-made abstractions
- Provide design alternatives
- Streamline communication among designers

More on Design Patterns

- Given common names
  - e.g. “Bridge” – builds an interface and an implementation in such a way that either can vary without the other varying
- Could go into much more on this

Other Heuristics

- Strong Cohesion
  - All routines support the main purpose
- Build Hierarchies
  - Manage complexity by pushing details away
- Formalize Class Contracts
  - Clearly specify what is needed/provided
- Assign Responsibilities
  - Ask what each object should be responsible for

More Heuristics

- Design for Test
  - Consider how you will test it from the start
- Avoid Failure
  - Think of ways it could fail
- Choose Binding Time Consciously
  - When should you set values to variables
- Make Central Points of Control
  - Fewer places to look -> easier changes
More Heuristics

• Consider Using Brute Force
  − Especially for early iteration
  − Working is better than non-working
• Draw Diagrams
• Keep Design Modular
  − Black Boxes

Design Practices
(we may return to these)

• Iterate – Select the best of several attempts
• Decompose in several different ways
• Top Down vs. Bottom Up
• Prototype
• Collaborate: Have others review your design either formally or informally
• Design until implementation seems obvious
  − Balance between “Too Much” and “Not Enough”
• Capture Design Work
  − Design documents