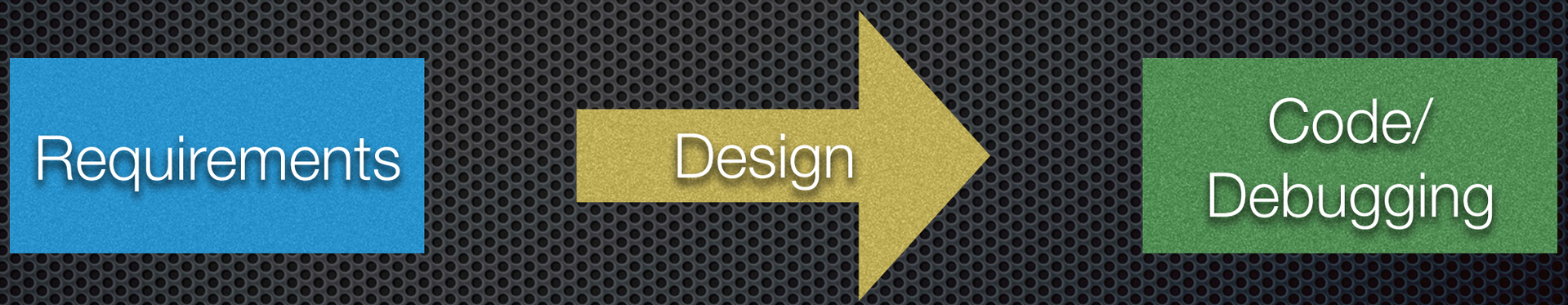
Design Considerations

CAC 430

Design Challenges

- “Software Design” - the conception, invention, or contrivance of a scheme for turning a specification for computer software into operational software



Design is Wicked

https://www.kiro7.com/news/tacoma-narrows-bridge-collapses-on-november-7-1940/1006176170/

- A problem that could be clearly defined only by solving it, or by solving part of it
- Defined by Horst Rittel (design theorist and professor) and Melvin Webber (urban designer and theorist) in 1973
- Have to “solve” the problem once to clearly understand it, and then solve it again to create the optimal solution

Design is Sloppy

- Going to make mistakes
- Try something, fail, try again
- How much is enough?
- What is enough detail?
- “When you’re out of time”

Design...

- is about tradeoffs and priorities: many competing characteristics, what is the highest priority? (ex: response time, developer time, etc.)
- involves restrictions: want to create possibilities but also want to restrict possibilities
- is nondeterministic: no one “right” solution
- is a heuristic process: involves trial and error; no tool is right for everything - design techniques tend to be “rules of thumb” or “things to try that sometimes work”
- is emergent: evolve and improve through design reviews, informal discussions, experience writing and revising code

Key Design Concepts

1. Complexity
2. Desirable Characteristics of Design
3. Levels of Design

Managing Complexity

- Two classes of problems: essential and accidental
- Brooks refers to philosophical definition - the essential properties are the properties that a thing must have in order to be that thing
- Accidental properties are the properties a thing just happens to have, properties that don’t really bear on whether the thing is what it is

https://www.motorauthority.com/news/1126971_2020-jeep-wrangler-jpp-20-is-a-mopar-made-off-roader-for-after-the-apocalypse

- Brooks: major accidental difficulties in software were addressed long ago
  - Difficult language syntax: assembly to 3rd generation
  - Noninteractive computers: time-share OS to batch-mode systems
- What about the essential difficulties? The essential is what connects the software to the real-world.
  - Slow process or fast?

Importance of Managing Complexity

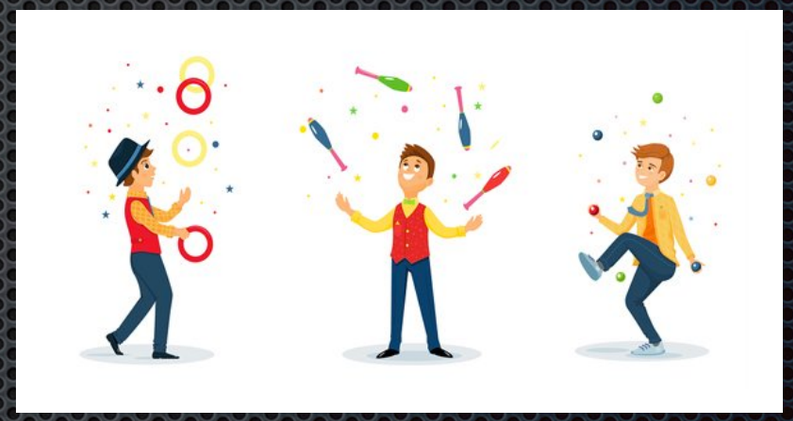
- “There are two ways of constructing a software design: one way is to make it so simple that there are *obviously* no deficiencies, and the other is to make it so complicated that there are no *obvious* deficiencies.” - C. A. R. Hoare
- What do you think is the primary reason projects fail?

Importance of Managing Complexity

- “There are two ways of constructing a software design: one way is to make it so simple that there are *obviously* no deficiencies, and the other is to make it so complicated that there are no *obvious* deficiencies.” - C. A. R. Hoare
- What do you think is the primary reason projects fail?
  - When it’s technical, it’s usually due to uncontrolled complexity

- Dijkstra pointed out that computing is the only profession in which a single mind is obliged to span the distance from a bit to a few hundred megabytes, a ratio of 1 to 10^9, or nine orders of magnitude

- He also mentioned that no one's skull is really big enough to contain a modern computer program - focus on one part at a time



https://www.jugglingtricks.net/what-is-juggling/

- At the architecture level, we break systems into subsystems - easier to focus on several simple pieces of information that one complex piece of information

- The more independent the pieces are, the safer it is to focus on one thing at a time; **separation of concerns**

“One symptom that you have bogged down in complexity overload is when you find yourself doggedly applying a method that is clearly irrelevant, at least to any outside observer. It is like the mechanically inept person whose car breaks down - so he puts water in the battery and empties the ashtrays.”

–P. J. Plauger

Attacking Complexity

- Three causes of costly, ineffective designs:

1. complex solution to a simple problem
2. simple, incorrect solution to a complex problem
3. inappropriate, complex solution to a complex problem

- Solution:
  - minimize the amount of essential complexity that anyone's brain has to deal with at any one time
  - keep accidental complexity from needlessly proliferating

Desirable Characteristics of Design

- Many competing goals
- Some characteristics are just general qualities for a good program: reliability, performance, etc.
- Others are internal characteristics of design…
  - **Minimal complexity:** avoid “clever”, aim for “simple”/“easy-to-understand” - If your design doesn’t let you safely ignore most other parts of the program when you’re immersed in one specific part, the design isn’t doing its job

- **Ease of maintenance:** design for the maintenance programmer, this is your audience, system should be self-explanatory

- **Loose coupling:** designing so that you hold connections among different parts of a program to a minimum; use principles of good abstractions in class interfaces, encapsulation, and information hiding to design classes with as few interconnections as possible - this minimizes work during integration, testing, and maintenance

- **Extensibility:** can enhance a system without causing violence to the underlying structure (open-closed principle - open for extension, closed for modification)

- **Reusability:** designing the system so that you can reuse pieces of it in other systems
- **High fan-in:** having a high number of classes that use a given class; implies that a system has been designed to make good use of utility classes at the lower levels in the system
- **Low-to-medium fan-out:** having a given class use a low-to-medium number of other classes (typically 7 is too much); also applicable to methods
- **Portability:** designing the system so you can easily move it to another environment

- **Leanness:** designing the system so that it has no extra parts
  - Voltaire - a book is finished not when nothing more can be added but when nothing more can be taken away
  - Software - all code has to be developed, reviewed, tested, and considered when code is modified; future versions must be backward compatible
    - Dangerous question - “It’s easy, so what will we hurt by putting it in?”
- **Stratification:** trying to keep the levels of decomposition stratified so that you can view the system at any single level and get a consistent view (ex: Interface)
- **Standard techniques:** using standardized, common approaches makes the system familiar

Levels of Design

1. Software System
2. Division into subsystem/packages
3. Division into classes within packages
4. Division into data and routines within classes
5. Internal routine design

Software System

- Entire system
- Don’t jump from the system level into designing classes

Division into Subsystems or Packages

- Identification of all major subsystems
  - Database
  - User interface
  - Business rules
  - Command interpreter
  - Report engine
- Decide how to partition the program into major subsystems and define how each subsystem is allowed to use each other subsystem

User Interface

Graphics

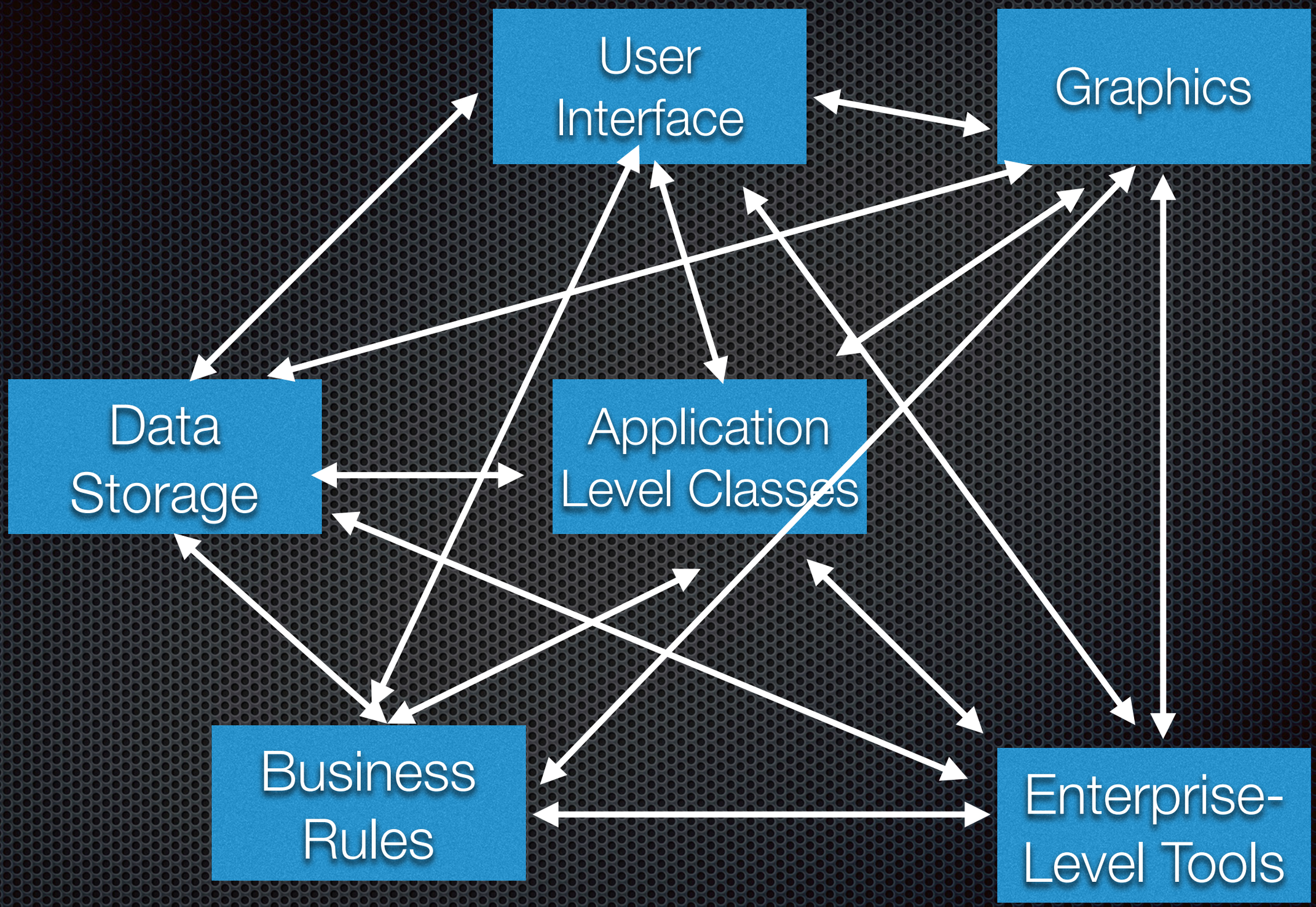
Data Storage

Application Level Classes

Business Rules

Enterprise-Level Tools

What's the issue?



- How many different parts of the system does a developer need to understand at least a little bit to change something in the graphics subsystem?
- What happens when you try to use the business rules in another system?
- What happens when you want to put a new user interface on the system, perhaps a command-line UI for test purposes?
- What happens when you want to put data storage on a remote machine?

https://www.familyhandyman.com/project/how-to-fix-a-leaking-sink-sprayer/

- Keep connections easy to understand
- Best to restrict communications in the beginning and relax later if needed
- The simplest relationship is to have one subsystem call routines in another
- A more involved relationship is to have one subsystem contain classes from another
- The most involved is to have classes in one subsystem inherit from classes in another
- **General Rule: system-level diagram should be an acyclic graph - no circular relationships**

Division into Classes

- Details of the ways in which each class interacts with the rest of the system are also specified as the classes are specified
- The class's interface is defined
- Primarily want to make sure all the subsystems have been decomposed with enough detail to implement the parts

Division into Routines

- Divide each class into routines
- Often results in a better understanding of the class's interface, and that causes corresponding changes to the interface
- Level of decomposition often left up to the individual programmer

Internal Routine Design

- Laying out the detailed functionality of the individual routines
- Typically left to the individual programmer
  - writing pseudocode
  - looking up algorithms in reference books
  - deciding how to organize the paragraphs of code in a routine
  - writing code

Design Building Blocks: Heuristics

- How do you feel when code doesn't work the way you think it should?
- We like deterministic behavior
- Because design is nondeterministic, application of an effective set of heuristics is the core activity in good software design

Find Real-World Objects

- “Ask not first what the system does; ask WHAT it does it to!” - Bertrand Meyer
- OOP approach
  - Identify the objects and their attributes (methods and data)
  - 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-which parts will be public/private
  - Define each object’s public interface

Let’s think about a time-billing system

- We might have…
  - Client
  - Employee
  - Timecard
  - Bill
- 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
- Define each object’s interface

References

- McConnell, Steve. (2004). Code Complete: A Practical Handbook of Software Construction, 2nd Edition. Microsoft.