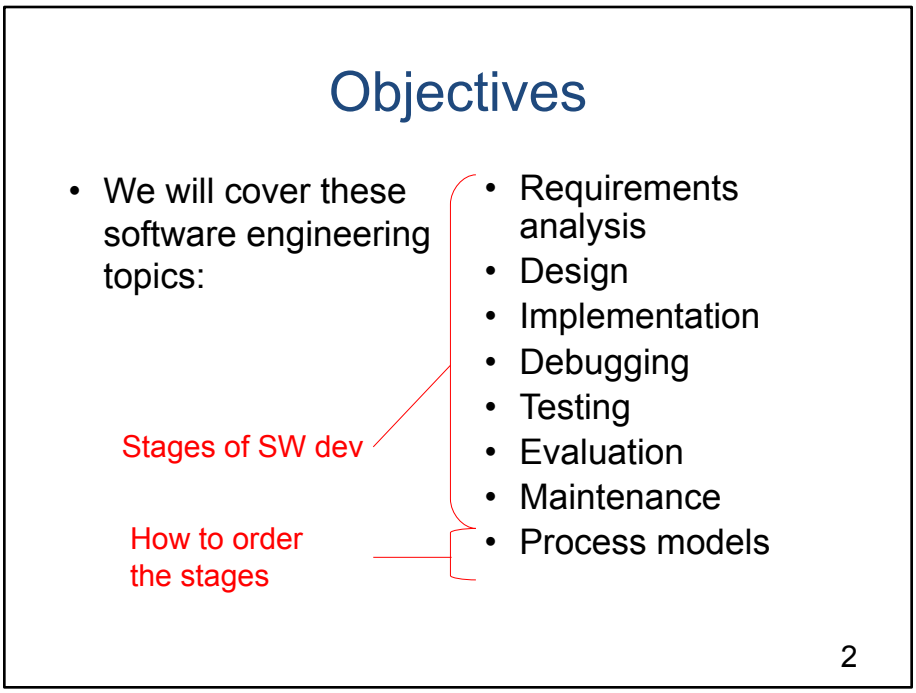
Software Engineering (Part 1)

Robert M. Dondero, Ph.D.
Princeton University



Objectives

[see slide]

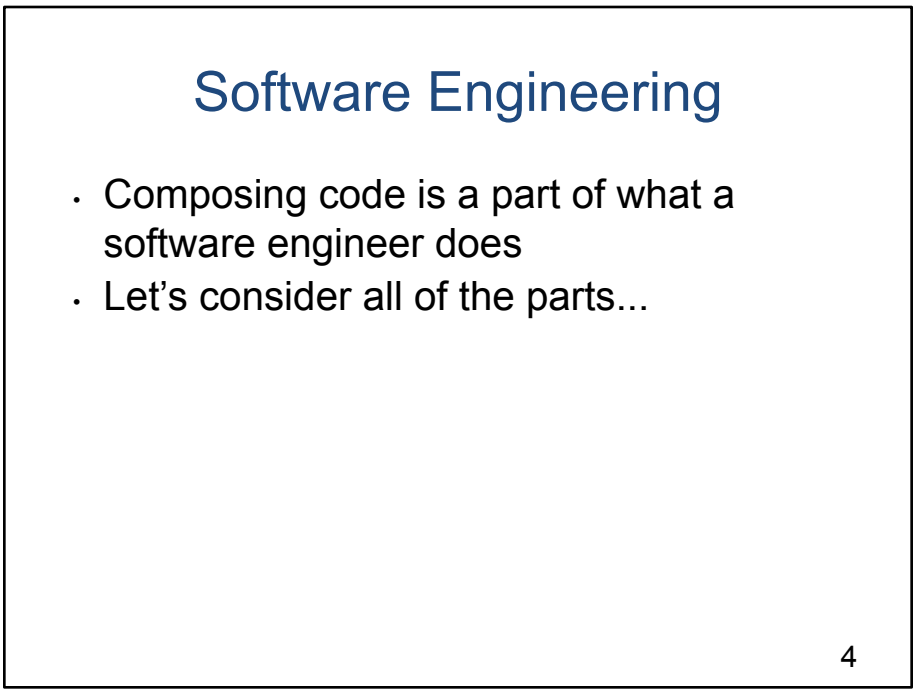
In other words...
Describe some software engineering techniques and tools that:

- You **might have used in** your COS 333 project
- You **might still use in** your COS 333 project
- You **might use beyond** your COS 333 project

Objectives

Software Engineering lecture slide decks:

Part 1	**Requirements analysis** **Design (general)**
Part 2	Design (object-oriented) Implementation Debugging
Part 3	Testing Evaluation
Part 4	Maintenance Process models



Software Engineering

Composing code is a part of what a software engineer does
The other parts are important, but rarely are covered in academia
Let's consider all of the parts...

You've decided to create a software system. What's your first step?

Agenda

- **Requirements analysis**
- Design
- Implementation
- Debugging
- Testing
- Evaluation
- Maintenance
- Process models

Requirements Analysis

- ***Requirements analysis***
  - **Who** are the system's users?
  - **What** should the system do to fulfill the users' needs?

What kinds of requirements should you gather?

Requirements Analysis: Kinds

- Always:
  - ***Functional*** requirements
- Sometimes:
  - ***Data*** requirements
  - ***Environmental*** requirements
  - ***User*** requirements
  - ***Usability*** requirements

Yvonne Rogers, Helen Sharp, Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction (3rd Edition)*. Wiley, 2011.

9

Requirements Analysis: Kinds

Functional requirements
- What must the system do?

Data requirements
- How much data must the system store?
- How long must the data persist?
  - Must the data be archived, maybe for legal reasons?
- How accurate must the data be?
- How volatile will the data be?

Environmental requirements
- What will be the system's…
  - Physical env (lighting, noise, dust, protective gloves,...)
  - Social env (collaboration, coordination, synchronous, …)
  - Organizational env (support, …)
  - Technical env (power, compatibilities, …)
- Especially important with the rising popularity of mobile devices

User requirements
- Will the system's users be novices or experts **in the domain**?
- Will the system's users be novices or experts **with computers**?
- How many users will there be?
  - How many distinct users will there be?
  - How often will each user use the system?
  - Many users, infrequent use => learnability is important

Few users, frequent use => learnability isn't as important; ease of use is critical

Usability requirements

How easy must the system be to **use**?
How easy must the system be to **learn**?
How **memorable** must the system be?
How much **fun** must the system be to use?
How **motivating** must the system be?

Examples:

Developing business apps => consider functional requirements
Developing a computer system to be used by a Navy pilot => consider may other kinds of requirements

Important information must be conveyed by both sight and sound

How should you go about gathering those requirements?

Requirements Analysis: Gathering

- Questionnaires
- Interviews
- Focus groups
- Direct observation
- Studying documentation
- Researching similar products

Users visit the pgmmers

Pgmmers visit the users

Yvonne Rogers, Helen Sharp, Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction (3rd Edition)*. Wiley, 2011.

How should you structure the requirements that you've gathered?

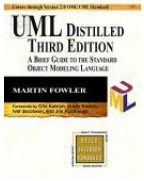
Requirements Analysis: Structuring

- Create models of the user's domain
  - A popular set of modeling notations...
  - ***Unified Modeling Language (UML)***

Requirements Analysis: Structuring

- ***Unified Modeling Language (UML)***
  - **Who**: Grady Booch, James Rumbaugh, Ivar Jacobson
  - **When**: 1980s
  - **What**: A set of notations
  - **Why**: To model user domains and systems



"The three amigos"

Requirements Analysis: Structuring

Unified Modeling Language (UML)

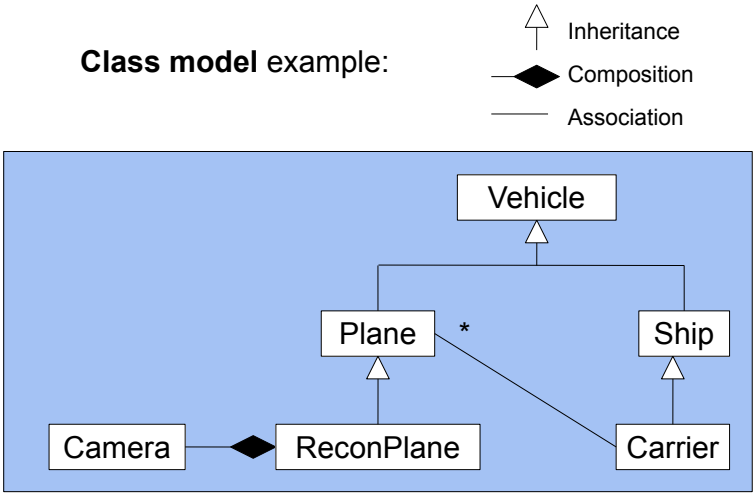
Who: Grady Booch, James Rumbaugh, Ivar Jacobson
"The three amigos"
When: 1980s
What: A set of notations
Why: To model user domains and systems

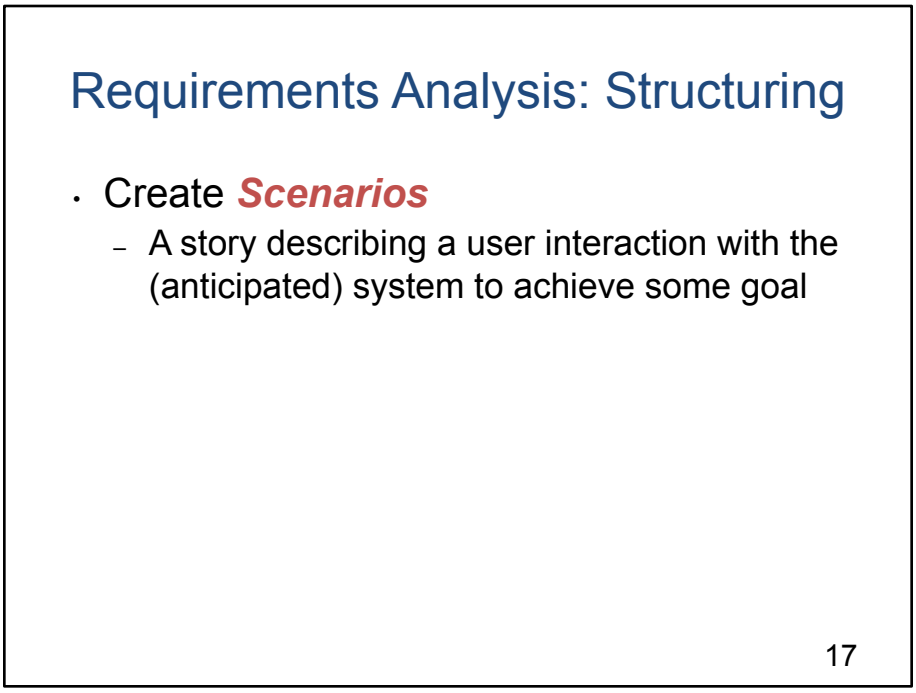
Requirements Analysis: Structuring

- Create ***Class Model(s)***
  - A UML notation
  - Describes classes of objects in the user's domain
  - Does not (necessarily) refer to the system to be built

Requirements Analysis: Structuring

Class model example:





Requirements Analysis: Structuring

Create ***Scenarios***

A story describing a user interaction with the (anticipated) system to achieve some goal

In one scenario the user creates a new account and logs in

In another scenario a user searches for something

…

Requirements Analysis: Structuring

- Create ***Prototype(s)***
  - Bridge from requirements analysis to design
  - Low-fidelity
  - High-fidelity

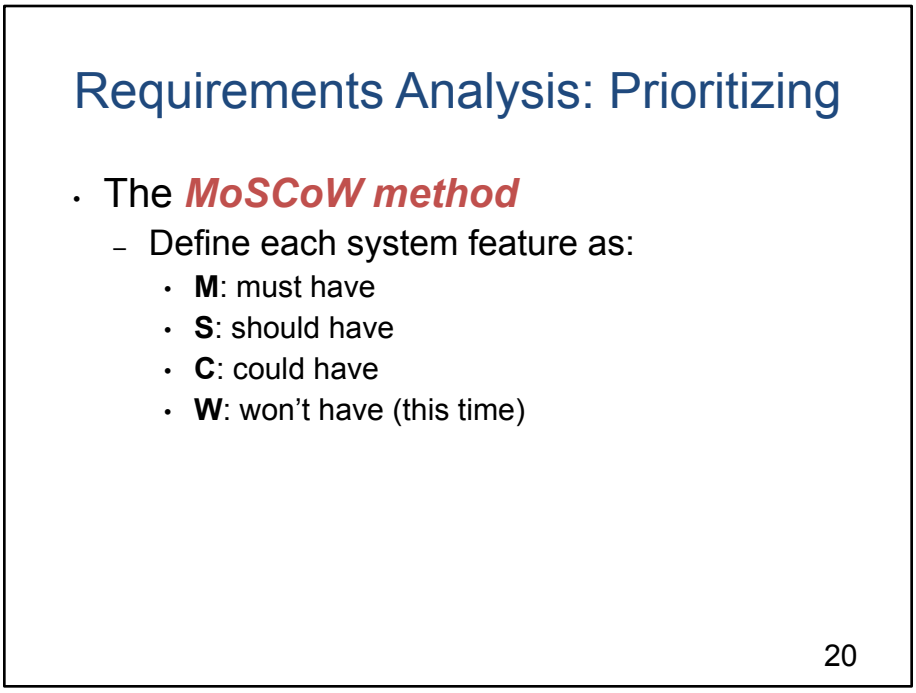
18

Requirements Analysis: Structuring

Create ***Prototype(s)***

- Bridge from requirements analysis to design
- Low-fidelity
  - Storyboards, index cards, …
  - Supporting tools: Figma, Draw.io, Framer...
- High-fidelity
  - Shallow HTML docs, “Wizard of Oz”, …
  - System with only the most common logical paths implemented

You probably can't fulfill all of the user's requirements. And you certainly can't fulfill all of the user's requirements right away. How should you prioritize the requirements?



Requirements Analysis: Prioritizing

The ***MoSCoW method***

Define each system feature as:

M: must have: the feature is required

S: should have: the feature is important

C: could have: the feature is desirable

W: won't have (this time): the feature is optional

Requirements Analysis: Conclusion

- In the **academic** world:
  - Student programmers often are given requirements
- In the "**real**" world:
  - (Senior) programmers often must know how to **gather**, **structure**, and **prioritize** requirements

You've gathered, structured, and prioritized the system's requirements. What should you do next?

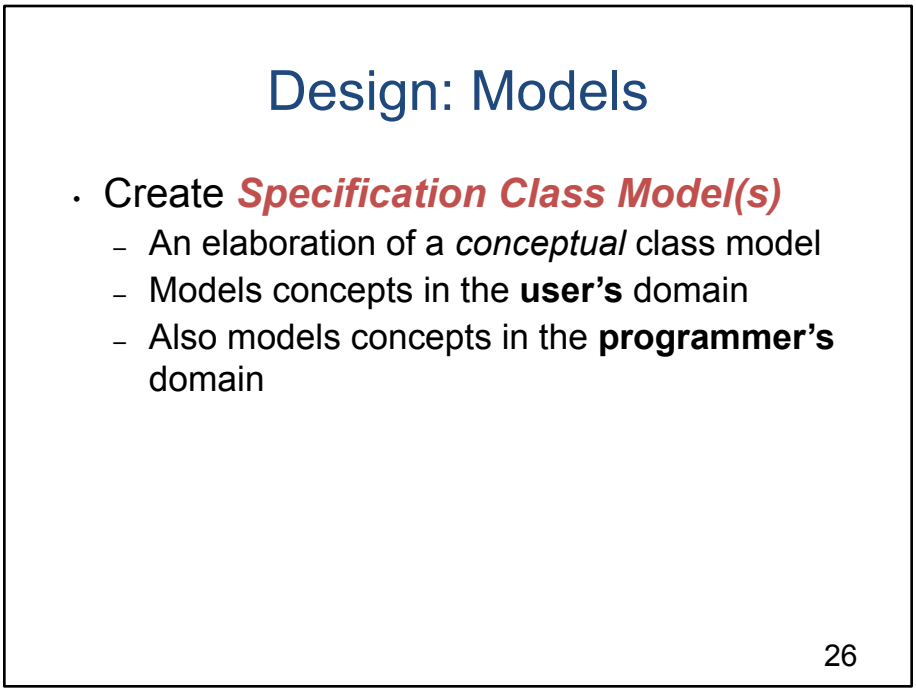
Agenda

- Requirements analysis
- **Design**
- Implementation
- Debugging
- Testing
- Evaluation
- Maintenance
- Process models

Design

- ***Design***
  - **How** should the system work?

How should you specify the system's design?



Design: Models

Create ***Specification Class Model(s)***

A *specification* class model…

Is an elaboration of a *conceptual* class model

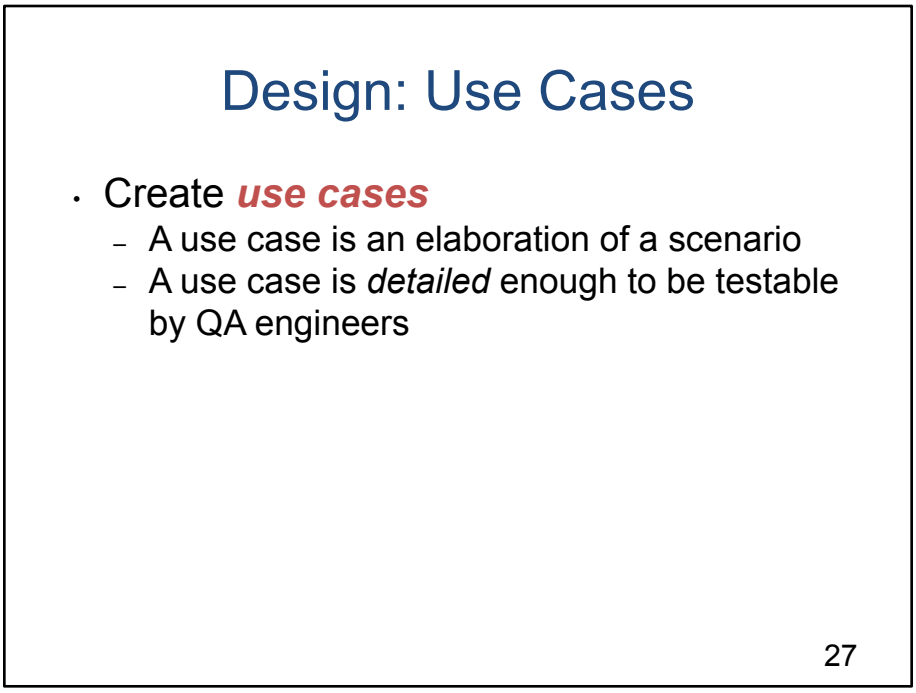
Models concepts in the **user's** domain

Vehicles, Ships, …

Also models concepts in the **programmer's** domain

Fields, methods, …

List objects, Dictionary objects, Thread objects, Database objects, ...



Design: Use Cases

Create ***use cases***

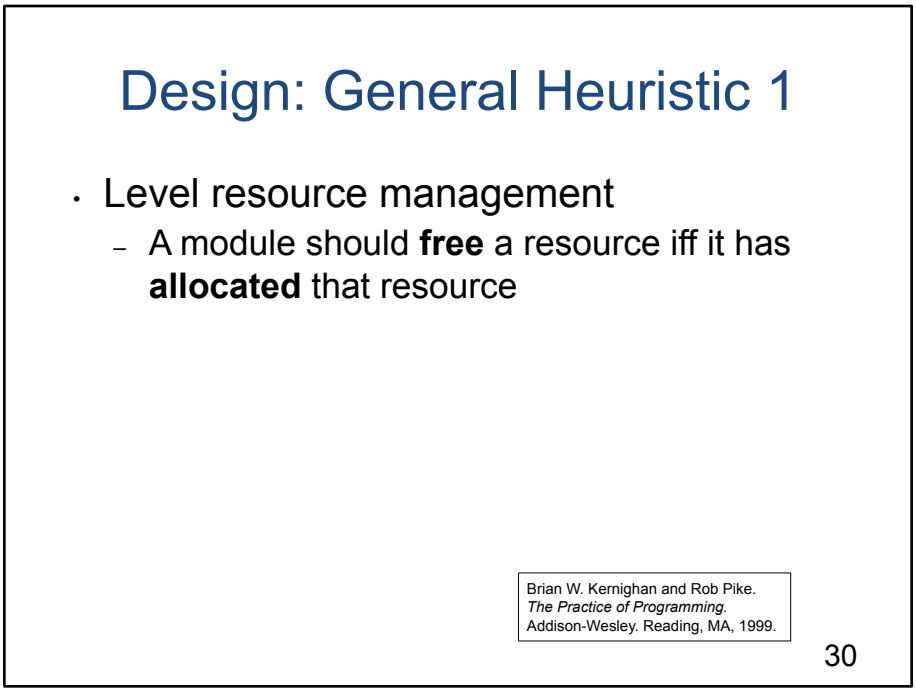
A use case is an elaboration of a scenario
A **scenario** corresponds to multiple **use cases**
Each uses case describes the *detailed* steps that a user must take to complete part of the scenario
A use case is *detailed* enough to be testable by QA engineers

In your project Users Guide we'll count on you giving us detailed use cases that guide us through all/most feature of your application

What heuristics should you keep in mind when designing the system?

Design: Heuristics

- Use **design heuristics**
  - Some are general
  - Some are specific to OO pgmming



Design: General Heuristic 1

Level resource management

A module should **free** a resource iff it has **allocated** that resource

Resources: memory, file handles, …

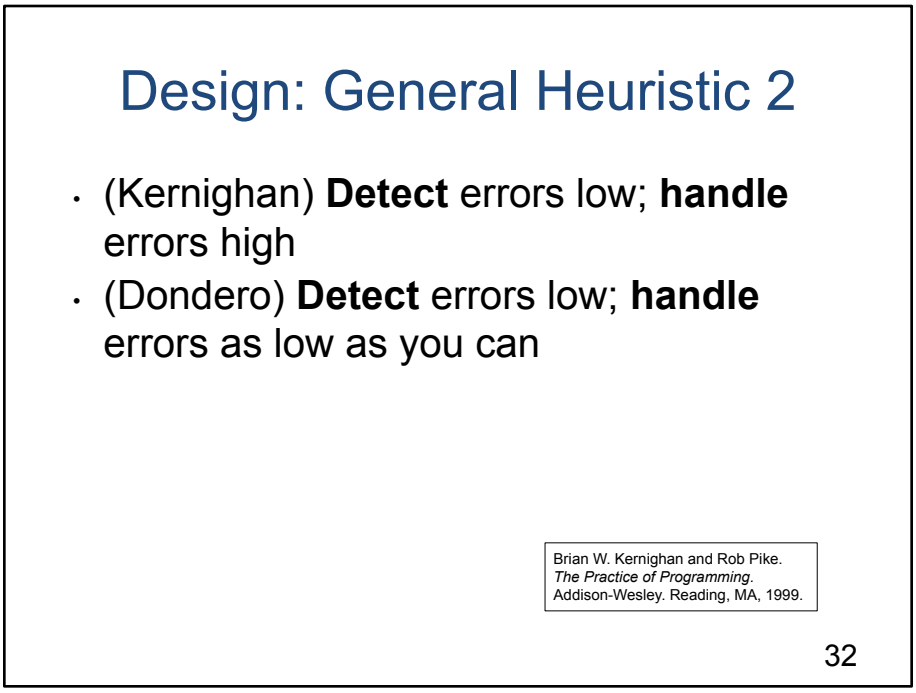
Document all violations clearly

Example (COS 217 symtable module):

Symtable_new() allocates memory for Symtable object
SymTable_put() allocates memory for a binding and a key
SymTable_remove() should free memory for binding and a key
SymTable_free() should free memory for all bindings, all keys, and the Symtable object itself
Since the SymTable module allocates memory, it should free that memory

Example (Penny)

database.py module opens a DB connection, so it also closes the connection.



Design: General Heuristic 2

(Kernighan) **Detect** errors low; **handle** errors high

(Dondero) **Detect** errors low; **handle** errors as low as you can

A module should:

Detect errors

Handle errors if it can; otherwise…

Report errors to its clients (typically by throwing exceptions)

Example (Penny and maybe your Registrar's Office apps):

database.py module detects errors (database cannot be opened, corrupted database)

database.py doesn't know how to handle those error

Command-line application: write an error message to stderr

Client-server application: pop up a dialog box

Web application: send an error page as the response

So database.py reports errors to its clients (by throwing exceptions)

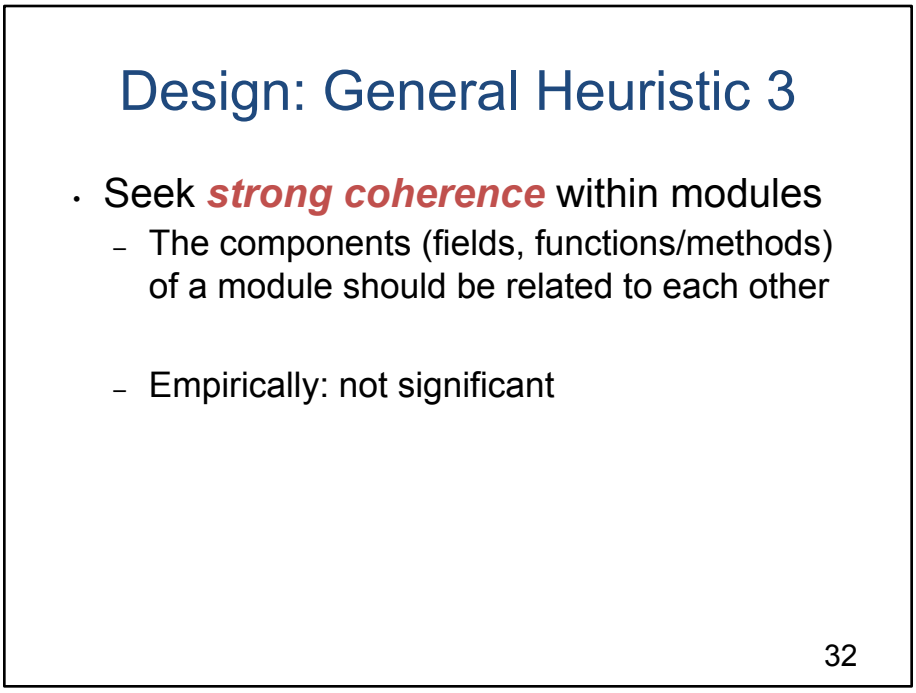
Generalizing:

Think of your application as a tree of modules; each parent module uses/calls down into its children

Draw a horizontal line, separating the program-dependent modules from the program-independent ones

Module above the line detects an error => it has the proper context to handle the error, and so should
Module below the line detects an error => it probably doesn't have the proper context to handle the error, and so should not

Instead it probably propagate the error upward (Python, Java, C++: should throw an exception)



Design: Generally Heuristic 3

[see slide]

Example (COS 217 str module):
str module contains functions related to string processing; it should contain no other kind of function

Programs that have weak coherence are not particularly buggy or difficult to maintain

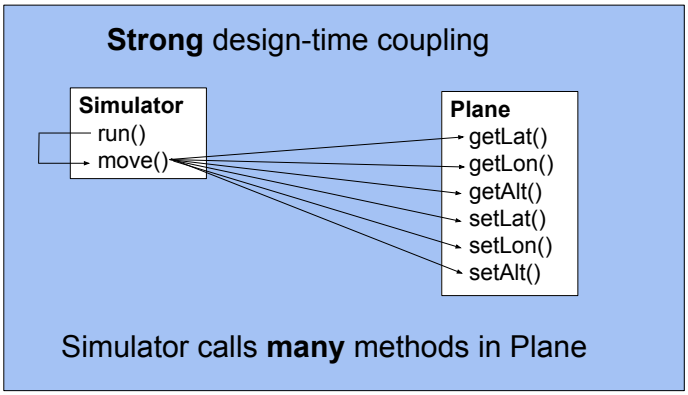
Design: General Heuristic 4

- Seek ***weak coupling*** among modules
  - Minimize interfaces
  - Encapsulate data
  - Empirically: **significant**

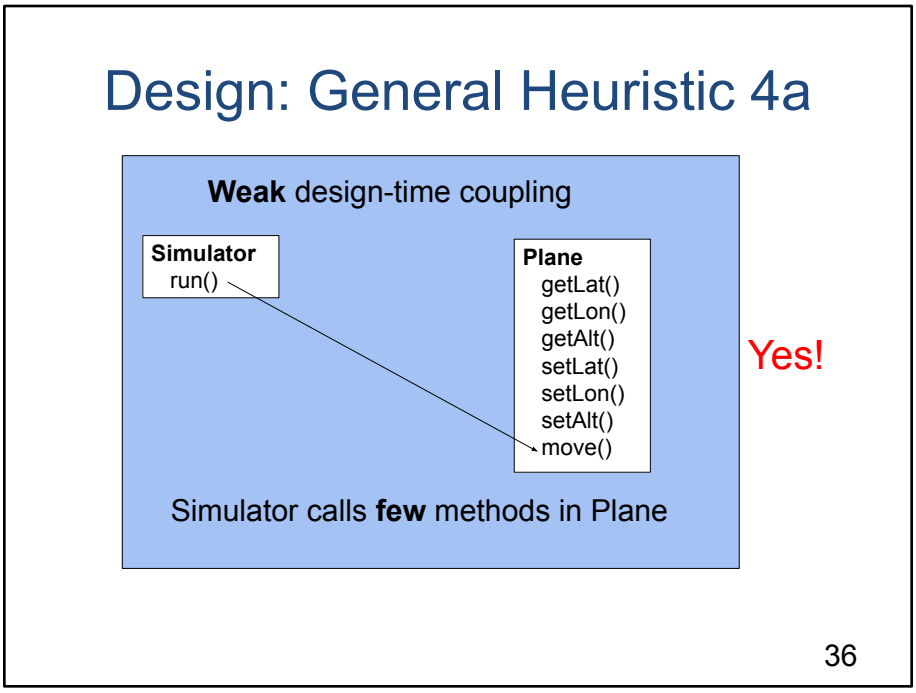
Design: General Heuristic 4a

- Seek weak ***design-time coupling***

Design: General Heuristic 4a



No!



Design: General Heuristic 4a

Example (COS 217 shell assignment)

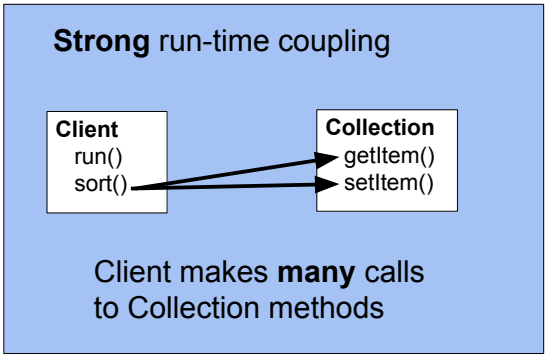
Example (Penny):

> Design 1:
> To perform a DB query, client of database.py calls one function to create a DB connection, one function to perform the query, one function to get the resulting rows and one function to close the DB connection
>
> Design 2:
> To perform a DB query, client of database.py calls one function which creates a DB connection, performs the query, closes the DB connection, and returns the resulting rows
>
> Design 2 has weaker run-time coupling
> Design 2 is better

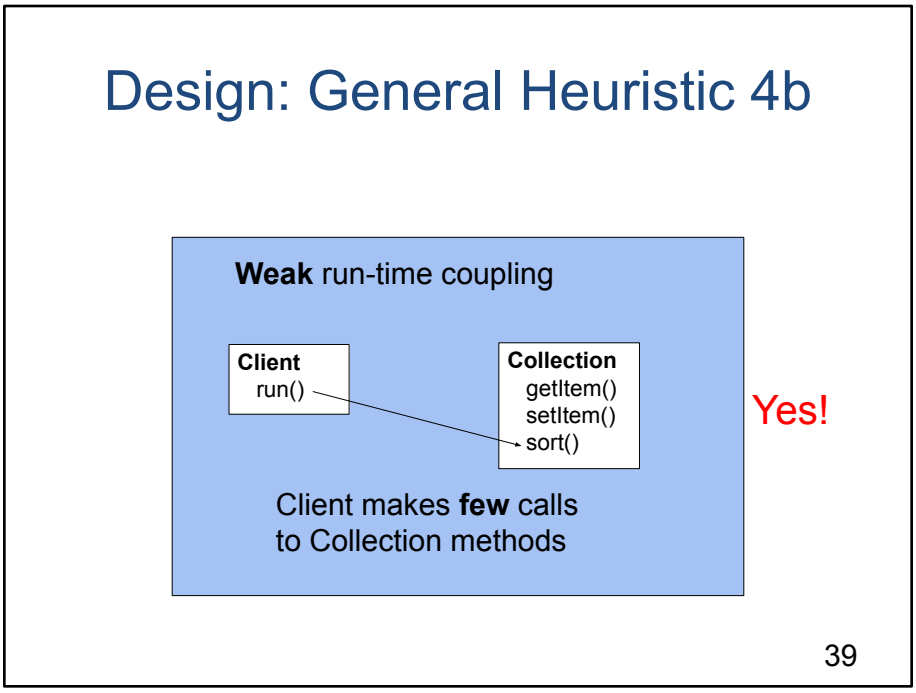
Design: General Heuristic 4b

- Seek weak ***run-time coupling***

Design: General Heuristic 4b



No!



Design: General Heuristic 4b

Example (Penny):

Design 1:
After a query, database.py module returns each resulting DB row one at a time
Client module makes **multiple** calls to database.py module to fetch all rows

Design 2:
After a query, database.py module returns all resulting DB rows in a list
Client module makes **one** call to database.py to fetch all rows

Design 2 has weaker run-time coupling
Design 2 is better

Continued in
Software Engineering (Part 2)…