CS-AD 209 - Spring 2013
Software Engineering
Lecture 3

Professor Michael Paik
mpaik@cs.nyu.edu

NYU

Attendance

Agenda

- Requirements
- Specifications
- Finite State Machines

Agenda

- Requirements
- Specifications
- Finite State Machines

NYU

Requirements

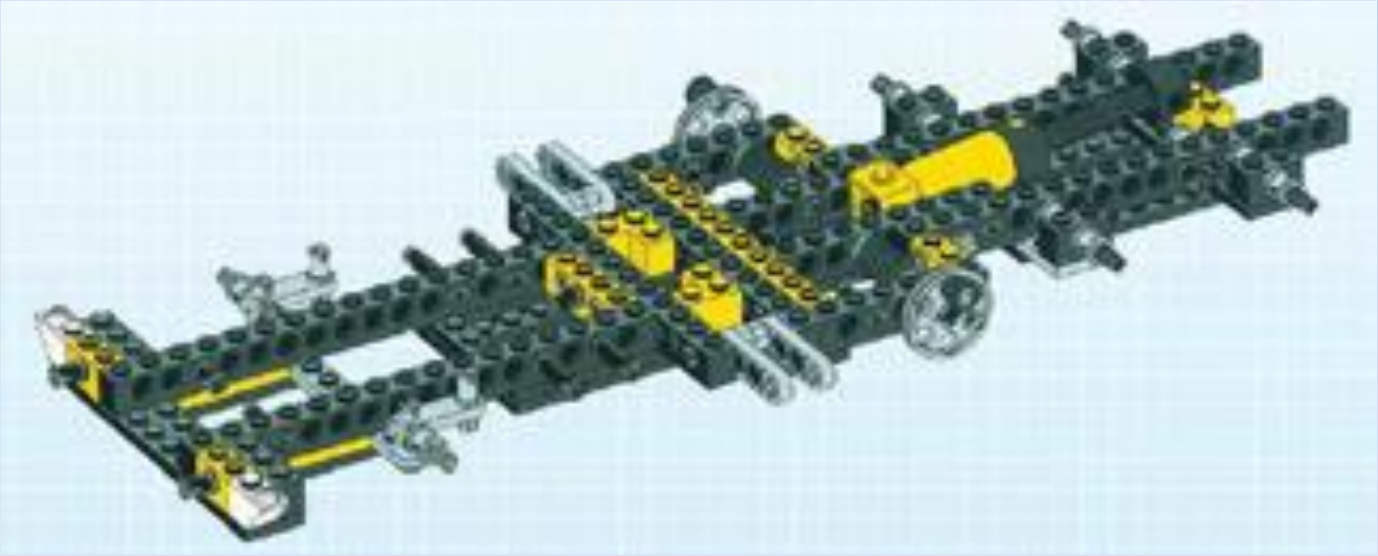
Requirements, or functional specifications, are the answers to '*what* does this do?'

This is not to be confused with '*how* does this do what it does?'

NYU

Requirements

'What does the finished product look like?'



Requirements

Requirements are how software engineers talk about user-facing software with business analysts, bizdevs, and other meatware

Requirements

This is not to say that requirements can be vague, imprecise, or aren’t important

Requirements are how you know when you’ve successfully completed your software engineering task

Requirements

Moreover, requirements help to limit scope/feature creep as time goes on

(More on this in the requirements gathering / NDD lecture)

Exercise

Let us say that a bizdev wants to create a Dropbox clone

What are the requirements?

Exercise

What types of words are we using to characterize requirements?

Requirements

Requirements generally fall into two categories:

1. Quantitative: how fast (performance), how many (scalability), how long (retention), etc.

2. Qualitative: features, usage scenarios (e.g. does this work offline?), etc.

In general it is the qualitative requirements that are hard to express and understand

Requirements

In short, requirements set the goal

We'll be talking about this further, this is just an apéritif

Agenda

- Requirements
- Specifications
- Finite State Machines

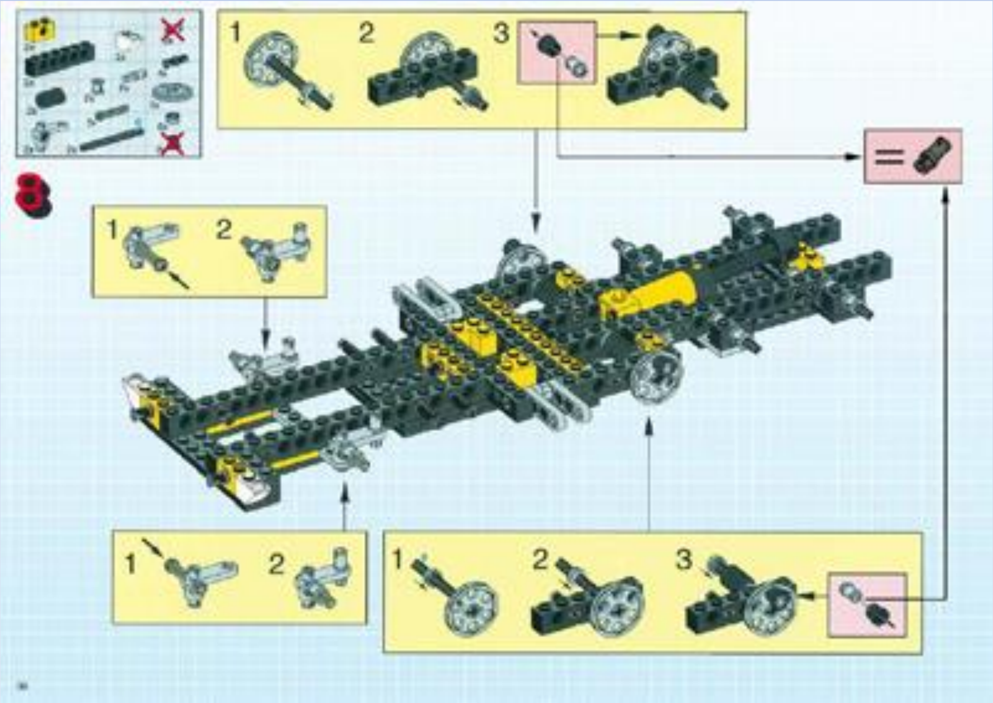
NYU

Specifications

In contrast to requirements, specifications express the *how* of the software project – how things are wired together and how they behave in each case when operating in tandem

Specifications

'How do the pieces fit together?'



Specifications

Specifications are harder to get right than you'd think and lead to some ugly failures:

1. 1992 – Bundestag sound system has requirement for zero feedback; software obeys and promptly automatically reduces volume to zero
2. 1995 – ESA Ariane 5 rocket reuses well-proven Ariane 4 guidance software; acceleration variable overflowed because the 5 accelerated much faster, and the rocket crashed
3. 1999 – NASA Mars Climate Orbiter becomes Mars Climate Atmospheric Fireball when ground station relaying data specifies in Imperial rather than Metric units
4. 1999 – NASA Mars Polar Lander crashes when leg deployment creates a momentary spike in a sensor reading, causing the software to think the lander had touched down and stop thrust

Groupthink: Answerphone

Let's try our hand at specifications, live

Groupthink: Answerphone

[Removed for publication]

You may find the Answerphone materials at Michael Ernst's Webpage

http://homes.cs.washington.edu/~mernst/

Specifications

Specifications fill in the gaps in requirements and detail specific interactions and behaviors

Just because something is not 'required' doesn't mean behavior should be unpredictable

Specifications are Hard

From a technical viewpoint they must be:

- Complete
  - is the last-number memory changed by `TALK` during a call?
- Consistent
  - what is displayed if there is an incoming call while you are reviewing answering machine messages?
- Precise
  - does `tones(string)` send tones to the audio bus?
- Concise
  - is it comprehensible?

Specifications are Hard

From a business viewpoint you have to compromise between what you *want* and what you *can have*

Specifications, Specifically

In practice, specifications for software also detail at least the methods that each class supports, and at an aggregate level, the API that a package/service presents to outside callers

This is necessary for 'blackbox' testing, which we'll come to a few lectures from now

Specifications, Specifically

Such specifications are usually in the form of *method contracts* (that is, what each method expects as input, and what the output is) of the type found in the Java API javadoc (e.g. http://docs.oracle.com/javase/7/docs/api/)

I won't go into this in detail as it's straightforward and I suspect you're all familiar with this type of specification, but it's important to note that Python, in contrast, does *not* provide such rigorous documentation in its API; you have to read the method descriptions to find out the types expected and returned

Agenda

- Requirements
- Specifications
- Finite State Machines

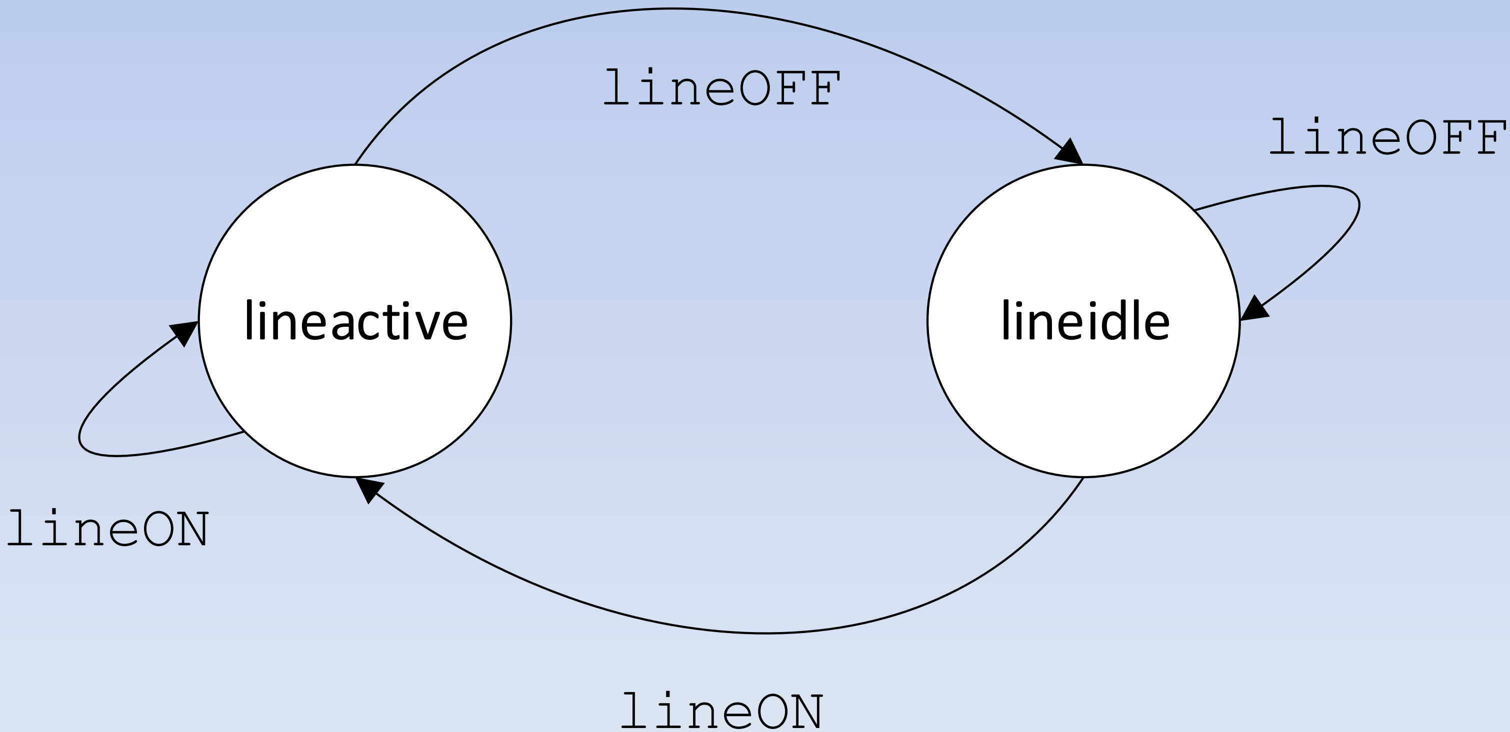
Finite State Machines

A Finite State Machine (FSM) (aka Finite State Automaton) describes the various *states* a system can be in, and the inputs to the system that cause transitions from one state to another

States are represented as circles or ellipses, transitions are represented as arrows, labeled with the transition-causing input

Toy Example: Phone Line

FSMs for small components are straightforward



Exercise

What does a plausible FSM for the Answerphone look like?

Finite State Machines

In order to be useful, the FSM must represent the state of the *entire* system, which can become unwieldy for large systems

The two usual approaches for large systems are either to create states representing all possible states of component parts (potentially huge number of states) or to maintain each component state separately and model inputs against a bus that interacts with all of them simultaneously

Advice

State requirements/specifications defensively – the people who are going to be working with your software are going to have heterogeneous skillsets and levels of competence

To make matters worse, there are the Richard Feynmans of the world

Fuzzy Green Balls

'I had a scheme, which I still use today when somebody is explaining something that I'm trying to understand: I keep making up examples. For instance, the mathematicians would come in with a terrific theorem, and they're all excited. As they're telling me the conditions of the theorem, I construct something which fits all the conditions. You know, you have a set (one ball)—disjoint (two balls). Then the balls turn colors, grow hairs, or whatever, in my head as they put more conditions on. Finally they state the theorem, which is some dumb thing about the ball which isn't true for my hairy green ball thing, so I say, "False!"'

Richard Feynman,
"Surely You're Joking, Mr. Feynman"

The 500-Foot View

The astute will observe that the previous lecture's coverage of MDDs lends itself to Models

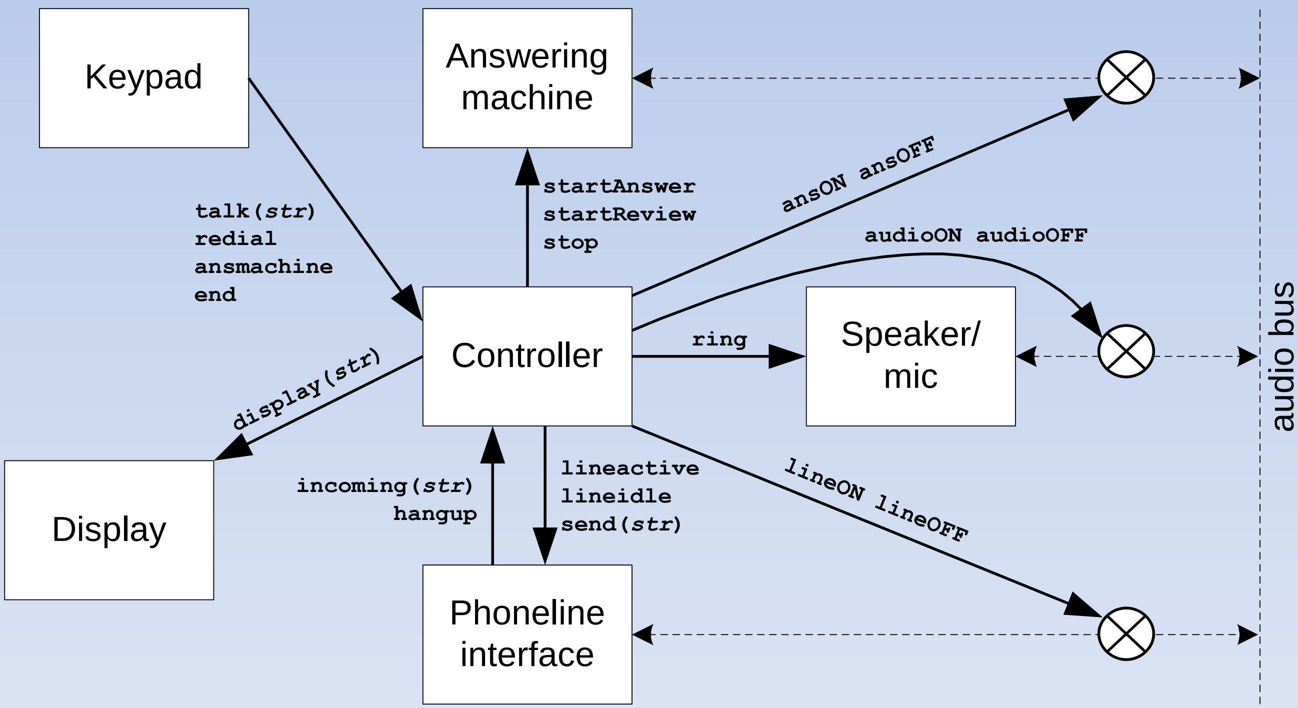
Similarly, today's material lends itself to Controllers

Every software artifact has at least these two-thirds of the Model-View-Controller paradigm (not all have views, e.g. APIs)

These ways of looking at software will continue to coalesce and reveal why some paradigms are paradigmatic

Homework

This is the system architecture for the Answerphone



NYU

Homework

The architecture diagram is messy =(

Assume that you have classes called `AudioBus`, `Display`, `PhoneLine`, `Keypad`, `Speaker`, `Mic`, `AnsweringMachine`, and `Controller`

Organize yourselves into two groups; each group will produce a logical MDD of the various components, a specification of the methods that each class supports including types, and produce a Finite State Machine describing the various transitions

Reading for Next Class

1. Chapters 1-2 (Introduction and Case Study) in the Gang of Four (Gamma et. al) Design Patterns book, and skim a few pattern descriptions in Chapter 3-5

2. (Optional, but enriching) http://blog.buildllc.com/2007/12/re-thinking-construction-documents/ discusses some of the design language used in construction documents, of which software design documents are a subset