Why are we here?

It’s harder to build solid software than it is to build a solid bridge
- Size and complexity makes it harder
- Teams make it harder (and easier)
- Customers make it harder
- Your project is hard

Software engineering is an answer
- Methods for tackling the problem
What is software engineering?

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

– IEEE Software Engineering Body of Knowledge

Engineering is:
The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property.

– Engineers Council for Professional Development

What is software engineering?

- It’s about communication and managing risk

www.projectcartoon.com
What is software engineering?

- It’s about communication and managing expectations

- It’s about being professional and delivering quality
Agenda

Processes
- Process models
- Process frameworks

Modeling
- The Unified Modeling Language

Activities
- Requirements
- Architecture
- Design

Process
- Organization of activities to achieve a certain objective

Defect Causal Analysis: Preventing Problems

Prioritize problem reports
- Showstoppers have top priority
- Perform AHP on remaining reports
- Create prioritized list of reports

Perform cause analysis
- Perform analysis in order of priority
- Create Ishikawa diagrams based on standard template
- Attach diagrams to the cause reports

Identify mitigations
- Identify common problem causes
- Prioritize causes in order of problem priority; common causes are given extra priority
- Identify actions in each category and attach to cause reports

Prioritize mitigations
- Estimate cost and benefit of mitigations
- Management decides which to implement based on costbenefit analysis
Lincoln Laboratory Process (1956)

Processes model

- Group of processes of the same character
The V model

Concept of operations → Validate concepts → Operations and maintenance

Requirements → Validate requirements, verify specification → Acceptance testing

System design → Verify system design → System testing

Program design → Verify module design → Integration testing

Implementation and unit testing

The waterfall model

System Requirements → Software Requirements → Analysis → Program Design → Coding → Testing → Operations

The waterfall model

![The waterfall model diagram]

Iterative development

![Iterative development diagram]
Spiral model


Process frameworks

- Somewhere between process models and processes

Unified Process
- Inception
- Transition
- Elaboration
- Construction

Rational Unified Process
Open Unified Process
Essential Unified Process

Scrum
- Scrum @ Medius
- Scrum @ Ericsson
- Scrum @ LiU-IT
Processes – conclusions

Process terminology
- Process model – the character of a process
- Process framework – a model that can be instantiated
- Process – a concrete organization of activities

More terminology
- Method – a concrete way of doing something
- Tool – a device that supports a given method

AGILE AND LEAN
Agile software development

- Reaction to traditional heavyweight methods
- Iterative with emphasis on reacting to **change**
- Rapid iteration, evolving requirements, emergent design
- Emphasis on collaboration in small cross-functional teams

Examples
- Extreme Programming
- Feature-Driven Development
- Open Unified Process

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.
Lean software development

- Complement to agile development methods
- Agile is about doing; lean adds a thinking dimension

- Lean eliminates waste – a little thinking can prevent redoing
- Lean brings decisions forward – but only the right ones
- Lean organizes teams – agile individuals within teams

- Lean is about high throughput (agile about low latency)
- Lean is about maximizing long-term business value

The Toyota Way
Some principles applied to software

- Eliminate waste through continuous improvement
  - Excess inventory, waiting, incorrect processing, defects

- Make decisions slowly, implement decisions rapidly
  - Go-and-see, determine cause, consider alternatives, consensus

- Build a culture of stopping to fix problems, to get quality right

- Use visual control to expose problems
  - Sort, straighten, shine, standardize, sustain

- Grow leaders who understand the work and live the philosophy
- Develop exceptional people and teams who follow the philosophy

REQUIREMENTS

Requirements engineering

- Domain understanding
- Requirements elicitation
- Requirements quality assurance
- Requirements evaluation
- Requirements specification and documentation
Requirements

- **What** is the problem to be solved
- **Why** does the problem need to be solved
- **Who** should be involved in solving it

- **How** should the problem be solved

Domain understanding and requirements elicitation

**Objectives**
- Understand the current situation
- Identify problems and opportunities
- Discover stakeholder needs
- Explore alternatives

**Knowledge acquisition**
- **Organization**: structure, objectives, policies, roles, responsibilities
- **Domain**: concepts, objectives, regulations, processes
- **Current system**: actors, tasks, workflows, problems, opportunities
Some methods

- Background study
- Talking to stakeholders
- Observation
- Questionnaires
- Storyboards
- Scenarios
- Mock-ups

Scenarios

- Early-stage requirements validation/elicitation technique
- **Narrative** of how things work or how they should work
  - **Passive**: stakeholders are told story – for **validation**
  - **Active**: stakeholders contribute to story – for **elicitation**
Scenario structure

- Who are the players
- What happens to them
- Why this happens
- What if some event occurs
- What could go wrong

A rescue worker reports an obstacle on a major road to the incident command system. The report is logged for postmortem analysis, and the obstacle is added to the overall situation report.

Since a new obstacle impacts operations, the operations section chief and liaison is notified of the new obstacle. Since it is a potential safety hazard, the safety officer is notified. Since the road is open to the public, the public information officer is notified of the new obstacle. Rescue workers known to be headed for the obstacle (based on their last known route and/or position) are also notified. If there is an area commander assigned to the location, that area commander is notified.

Mock-ups

**Functional prototypes**
- Not fully functional
- For validating functionality

**User interface prototype**
- Low or high fidelity possible
- To validate user interface
Requirements evaluation

Requirements issues
- Inconsistencies
- Conflicts
- Exposure to risk
- Prioritization
- Feasibility

Incident commander: all personal details of all potential victims should be available to all staff upon request.

Conflicts: this is illegal
Risk: exposure of information to outsiders
Feasibility: a lot of information is unavailable

Important techniques
- Conflict resolution
- Risk analysis (e.g. CORAS)
- Cost/value analysis (e.g. AHP)

Prioritization by cost/value

- Estimate (relative) cost
- Estimate (relative) values
Requirements specification

- Users need to make audio calls
- Communications should be secure

R3.2: Encryption
All network communications shall be encrypted with an AES128 session key.

R3.3: Session keys
Session keys shall be random and negotiated using a secure key exchange protocol.

R8.1 Audio calls
The user interface shall have a facility for initiating audio calls.

R8.6 Destination selection
When selecting a destination, common and important destinations shall be selectable with no more than two clicks.

R8.33 Voice encoding
Audio calls shall use G.729 encoding at 8kbit/s.

**Natural language**
- Easy to write and read
- Need good guidelines
- Risk of ambiguities

**Specification language**
- Precise and unambiguous
- Difficult to learn
- Good tool support

**S M A R T**
- Specific
- Measurable
- Attainable
- Realizable
- Time-bound or Traceable

**Functional vs. non-functional**

Modeling – a bit of both
S.M.A.R.T.  
Specific

Specific means
- There is no ambiguity
- Terminology is consistent
- Requirements are simple
- Appropriate level of detail

Consider
The map should be shown in a 640x480 window and obviously show most of the surrounding area including relevant map objects. Icons should be used to display map elements.

The map shall be shown in a 640px by 480px window.
The map shall show a user-selected area surrounding the user’s current position.
Relevant map objects (defined in R32) shall be displayed using icons (see R81) on the map.

S.M.A.R.T.  
Measurable

Measurable means
- It is possible to verify that this requirement has been met

Non-measurable means
- You never know if you’re done
- Opportunity for conflict

Consider
The program shall use as little memory as possible.
The program shall output the optimum route with respect to driving time.
The program shall never consume more than 48MB of memory.
S.M.A.R.T.

Attainable

Attainable means
- It is physically possible to meet the requirement under the given conditions

Consider
- The system shall be 100% reliable and 100% available.
- The system shall operate for at least 20 hours between charges.
- For any input, the system shall compute an optimal schedule in no more than one hour.

Ask
- Is there a theoretical solution
- Are there physical constraints
- Are there environmental constraints

Realizable

Realizable means
- It is possible to achieve this requirement given the constraints under which it is developed

Consider
- I have no good examples.
- Do you?

Ask questions like
- Can we do it?
- Is there enough time
- Are there enough people
- Is there enough money
S.M.A.R.T.
Traceable

Traceable means

- We can trace each requirement from analysis through design and implementation
- We can understand the reason for each requirement
- We can verify that each has been implemented
- Modifications are easy to make

Consider

- The program shall display time and date and the latest notifications and battery status and network status in the status bar.
- R23 The program shall display time and date in the status bar
- R24 The program shall display battery status in the status bar
- R25 The program shall display network status in the status bar

Discuss

The map should display the geographical location of the user.

The system shall never lose any data.

The user shall be able to make a voice call with good audio quality.

The system shall have a response time of one second.

The map display shall be 1024x768 pixels large.
Conclusions

On elicitation
- Crucial to understand the problem and solution domains
- Crucial to identify and communicate with all stakeholders

On evaluation
- Lots of techniques and tools exist for evaluation
- Don’t discard requirements – just defer them

On documentation
- If in doubt, write S.M.A.R.T. requirements
Models in software engineering

Documentation
- Concise and precise language
- Visual models easy to read

Verification
- Represent expected behavior
- Test behavior against model

Communication
- Models focus on key issues

Development
- Formal specification
- Stepwise refinement
- MDD/MDA

Exploration
- Test modifications on models

Model criteria

Mapping criterion
- There is an original object that is mapped to a model

Reduction criterion
- Not all properties of the original are mapped, but some are

Pragmatic criterion
- The model can replace the original for some purpose
The Unified Modeling Language (UML)

**Unified Modeling Language**
- Visual modeling language in software engineering
- Multiple levels of abstraction
- Well-defined metamodel
- Support for extensions

**Model**
- Concepts in the system
- Objects in the system
- Structures in the system

**Diagram**
- Visual representation of model

---

**UML Diagrams**

Diagram types include:
- Class Diagram
- Deployment Diagram
- Package Diagram
- Activity Diagram
- Interaction Diagram
- Use Case Diagram
- State Machine Diagram
- Sequence Diagram
- Communication Diagram
- Interaction Overview Diagram
- Timing Diagram

---
Class diagrams

Static structure

- Classes and their attributes
- Relationships between classes

Common elements

- Class and interface
- Attribute and operation
- Association and dependency
- Generalization and realization
- Note and constraint

Class diagram example
Component diagrams

Static structure
- Components and interfaces
- Relationships

Elements
- Components and ports
- Interfaces and collaborations
- Dependencies
- Notes and constraints

Component diagram example
Deployment diagrams

Run-time architecture

- Maps software elements to hardware elements (nodes)

Elements

- Nodes (various types)
- Artifacts and components
- Associations

Deployment diagram example
Activity diagram example

Use case diagrams
Conclusions

On modeling
- Models are not necessarily lean: what value do they contribute
- Models can support effective and efficient communications
- There are development methods that rely on models

On modeling languages
- The Unified Modeling Language is the most well-known language
- There are many others (not all are graphical)
Architecture

Architecture is design
Not all design is architecture

Software architecture
- Concentrates on issues that would be hard to change after the system is built
- Addresses non-functional and quality requirements
- Is a compressed view of a design

Software architecture goes beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

Garlan and Shaw: An Introduction to Software Architecture (1993)
Architecture

- Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.


- The software architecture of a program or a computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

  *Bass, Clements, Kazman: Software Architecture in Practice (2003)*

-------------------------

**Architecture (simplified)**

**Defines structure**
- System partitioning
- Component interfaces
- Component functionality

**Defines communication**
- Communications mechanisms
- Control flow
Some techniques for lean architecture

1. Separate components according to their rates of change
2. Partition the system so that each part can be managed autonomously
3. Let human considerations drive the partitioning
4. Don’t split a domain across architectural units or geographic locations
5. Module partitioning follows domain knowledge, if possible
6. Module partitioning follows end user cognitive model of the domain
7. Don’t forget system artifacts and other constraints
8. If you have a pattern language, then use it
Pushing my friends’ books 😊

Patterns

- A recurring solution to a recurring problem in context

Pattern Language
- A system of patterns for a particular domain
- The relationships and interactions between the patterns

A good pattern
- It solves a problem
- It is a proven concept
- The solution isn’t obvious
- It describes a relationship
- The pattern has a significant human component

*James Coplien*
Patterns

Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves. As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.

The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing.

– Christopher Alexander
The Timeless Way of Building

133 Staircase as a stage

If the entrances are in position – Main Entrance (110) – and the pattern of movement through the building is established – The Flow Through Rooms (131), Short Passages (132), the main stairs must be put in and given an appropriate social character. A staircase is not just a way of getting from one floor to another. The stair is itself a space, a volume, a part of the building; and unless this space is made to live, it will be a dead spot, and work to disconnect the building and to tear its processes apart. Therefore:

Place the main stair in a key position, central and visible. Treat the whole staircase as a room (or if it is outside, as a courtyard). Arrange it so that the stair and the room are one, with the stair coming down around one or two walls of the room. Flare out the bottom of the stair with open windows or balustrades and with wide-steps so that the people coming down the stair become part of the action in the room while they are on the stair, and so that people below will naturally use the stair for seats.
Architectural patterns in software

High-level patterns
- Layers
- Pipes and filters
- Blackboard

Distributed systems
- Microkernel
- Broker

Interactive systems
- Model-View-Controller (MVC)
- Presentation-Abstraction-Control

Layers

Context: A large system that requires decomposition

Problem: A mix of low- and high-level issues where high-level issues rely on the lower-level one.

Forces: Late source code changes, stable interfaces, exchangeable parts, subdivision of work

Solution: Structure the system in layers and place them on top of each other
Pipes and filters

Context: Processing data streams

Problem: Developing a monolithic module for processing may be infeasible due to complexity and inability to divide work.

Forces: Reuse of modules, reconfigurability, flexibility, varied input, output, data, parallelism.

Solution: Divide the system into several sequential processing steps.

Broker

Context: Distributed and possibly heterogeneous system with independent cooperating components.

Problem: When distributed components handle communication themselves, dependencies and limitations arise. Services for communication are also needed.

Forces: Transparent remote access, run-time reconfiguration, hide system details from users.

Solution: Introduce a broker component and have services register with the broker.
Model-View-Controller (MVC)

**Context:** Interactive applications with a flexible human-computer interface

**Problem:** Interfaces are prone to change and can be very complex.

**Forces:** Same information presented in multiple ways, display must reflect changes immediately, user interface should be easy to change.

**Solution:** Divide application into three areas: processing (model), output (view) and input (controller). Model notifies views of changes.

Architecture example redux

What pattern does this (mostly) conform to?
Conclusions

On architecture

- Architecture is the bridge from requirements to design
- Architecture defines the overall shape of the program
- Architecture determines many of its properties
- With emergent design, architecture is even more important
- There is no clear line between architecture and design

On patterns

- Don’t reinvent the wheel – if a pattern will do the job, use it
- If a pattern doesn’t do the job, invent something that will
Design

Continues from architecture
- Decomposition of the system
- Specification of interactions
- Decomposition of components

Agile development and design
- Design is emergent
- Architecture is critical

Traditional software design

Coupling
- Measure of interconnection among components

Cohesion
- Measure of how focused a single component is

Goals
- Low coupling
- High cohesion
Coupling

Loose (low) coupling
- Software easier to understand
- Easier to maintain, change, test

Types of coupling (low to high)
- Data coupling
- Stamp coupling
- Control coupling
- External coupling
- Common coupling
- Content coupling

Cohesion

Types of (low-medium) cohesion
- Logical cohesion
- Temporal cohesion
- Procedural cohesion
- Communicational cohesion

High (tight) cohesion
- Reduces coupling
- Easier to maintain, understand
Design patterns

A definition
- A recurring solution to a recurring design problem in context
- Discovered, not designed

Design patterns and agile
- Support emergent design
- Pattern language helps

Singleton (creation)

**Problem:** Need a single instance and don’t want to use global variables (or can’t)

**Solution:** Control creation of objects through static method

**Problems:**
- Similar to global variables
- Easy to overuse singletons

```python
class Singleton:
    def __init__(self):
        # Singleton instance initialization

    @staticmethod
    def get_instance():
        if not Singleton.instance:
            Singleton.instance = Singleton()
        return Singleton.instance
```

Don’t rely just on this book.
AbstractFactory (creation)

**Problem:** Select concrete types to create at run-time or easily change concrete types created.

**Solution:** Define an interface for creating instances, and create different implementations for different families of objects.

```
AbstractFactory
  + create_product() : AbstractProduct

ConcreteFactory
  + create_product() : Product
```

Composite (structural)

**Problem:** Want to represent part-whole relationships with a consistent interface.

**Solution:** Define a common component type and have both interior nodes and leaves implement the same interface. Dispatch operations from the composite to its components.

```
Component
  + addComponent(i: Component): void
  + removeComponent(i: Component): void
  + get_child(i: int) : Component
  + operation()

Leaf
  + operation()

Composite
  + addComponent(i: Component): void
  + removeComponent(i: Component): void
  + get_child(i: int) : Component
  + operation()

for c in self.children:
  c.operation()
```
Chain of responsibility (behavior)

**Problem:** Not sure what will handle a given event/command but know their relationships.

**Solution:** Create a chain of handlers, and just call the head of the chain with the command.

**Note:** Often used with composite and/or command patterns; the chain can be a tree if handlers dispatch to multiple objects or other chains.

```
class ChainOfResponsibility:
    def handle_request(self):
        self.successor.handle_request()

class ClientHandler:
    def handle_request(self):
        pass

class ConcreteHandler:
    def handle_request(self):
        self.successor.handle_request()

class ConcreteObserver:
    def update(self, state):
        for observer in self.observers:
            observer.notify()

class Subject:
    def __init__(self):
        self.state = State()
        self.subject = self
        self.observers = []

    def attach(self, observer):
        self.observers.append(observer)

    def detach(self, observer):
        self.observers.remove(observer)

    def notify(self):
        for observer in self.observers:
            observer.notify()

    def set_state(self, state):
        self.state = state
        self.notify()
```

Observer (behavior)

**Problem:** Keeping track of state

**Solution:** Automatically update dependents when object state changes

**Problems:**
- Garbage collection
- Infinite notify loops
- Error handling
Command (behavior)

Problem: Want to do undo, scripting, record commands, defer execution …

Solution: Encapsulate all state needed to perform command (or undo) in a command object, and invoke it using an invoker.

Conclusions

On design
- Design defines the detailed shape of the program
- Design determines many functional properties of the program
- Agile processes often omit the design phase

On patterns
- Don’t reinvent the wheel – if a pattern will do the job, use it
- If a pattern doesn’t do the job, invent something that will
- Beware of the design pattern hype!
WRAP-UP

Topics covered

Processes and modeling
- Processes models and frameworks
- Agile and lean development philosophies
- The Unified Modeling Language

Key software engineering activities
- Requirements
- Architecture and architectural patterns
- Design and design patterns
Other important topics

Software engineering activities
- More about requirements, design, architecture
- Coding and other implementation
- Testing, verification and validation
- Delivery, deployment and maintenance

Support activities
- Configuration management
- Documentation

In your project

Requirements – current step
- Determines the course of the project
- Read more about it – techniques for elicitation, analysis …

Architecture – next step
- Crucial – you are depending on emergent design
- Need architecture to support non-functional requirements

Design
- Emergent design – no up-front design needed