# Prototyping:
## Making design concepts real and testable

With contributions by: Steven Jacobs, Eck Doerry

With credit and thanks for some slides and ideas to Larry Bernstein's Quantitative Software Engineering Course at Stevens Institute of Technology, Hoboken, NJ

# Software Requirements Development Process

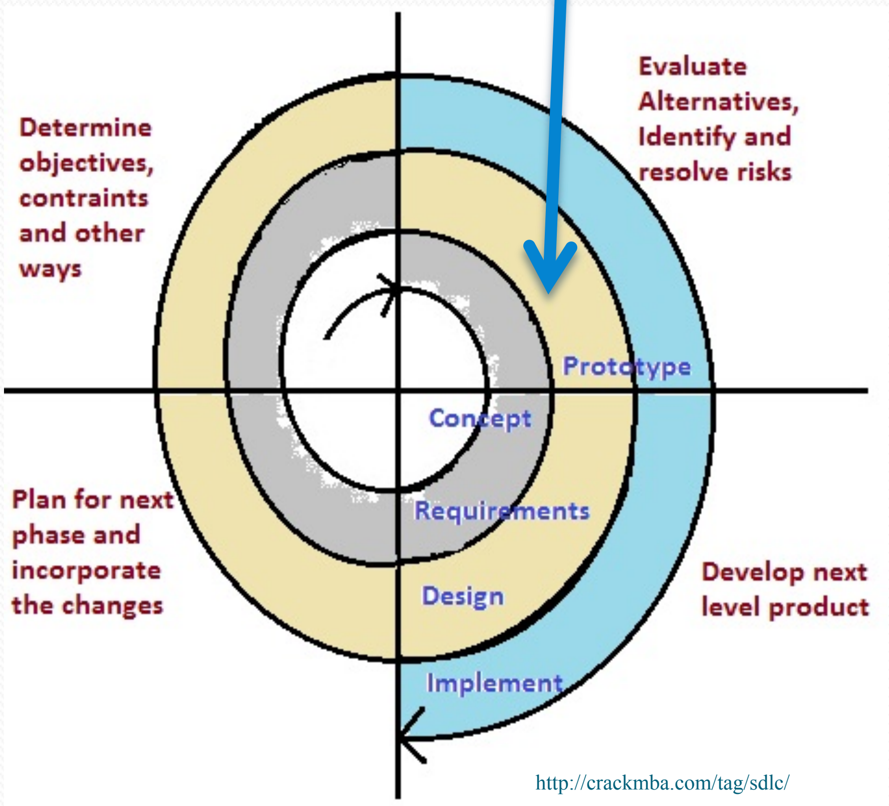
1. **Requirements Elicitation** (what the user wants)
2. **Requirements Analysis** (filling in the blanks)
3. **Use Cases** (Explore interaction dynamics)
4. ***Prototype/Modeling*** ***(prove that it can work)***
5. **Requirements Specification** (formalize it)
6. **Requirements Management** (trace/verify)

# Software Prototyping Process

- **Software prototyping:**
  - the process of creating an incomplete model of the future full-featured software
  - Or of key pieces/modules

- **Process:**
  - Identify basic requirements
  - Develop Initial prototype
  - Review with customers users
  - Revise and enhance

**Key tool for targeting implementation!**


Evaluate Alternatives, Identify and resolve risks
Determine objectives, contraints and other ways
Prototype
Concept
Requirements
Design
Implement
Plan for next phase and incorporate the changes
Develop next level product

http://crackmba.com/tag/sdlc/

# Prototypes as Specifications?

- Could we use a prototype as a specification?
  - "Let me show you sort of what I want (prototype)…"
  - "…and then we'll use that as spec!"

- No!  Because:
  - Some parts of the requirements may be impossible to prototype
    - E.g., safety-critical functions
  - An implementation has no legal standing as a contract!  It is no substitute for a clear specification of requirements.
  - Non-functional requirements cannot be adequately tested in a system prototype

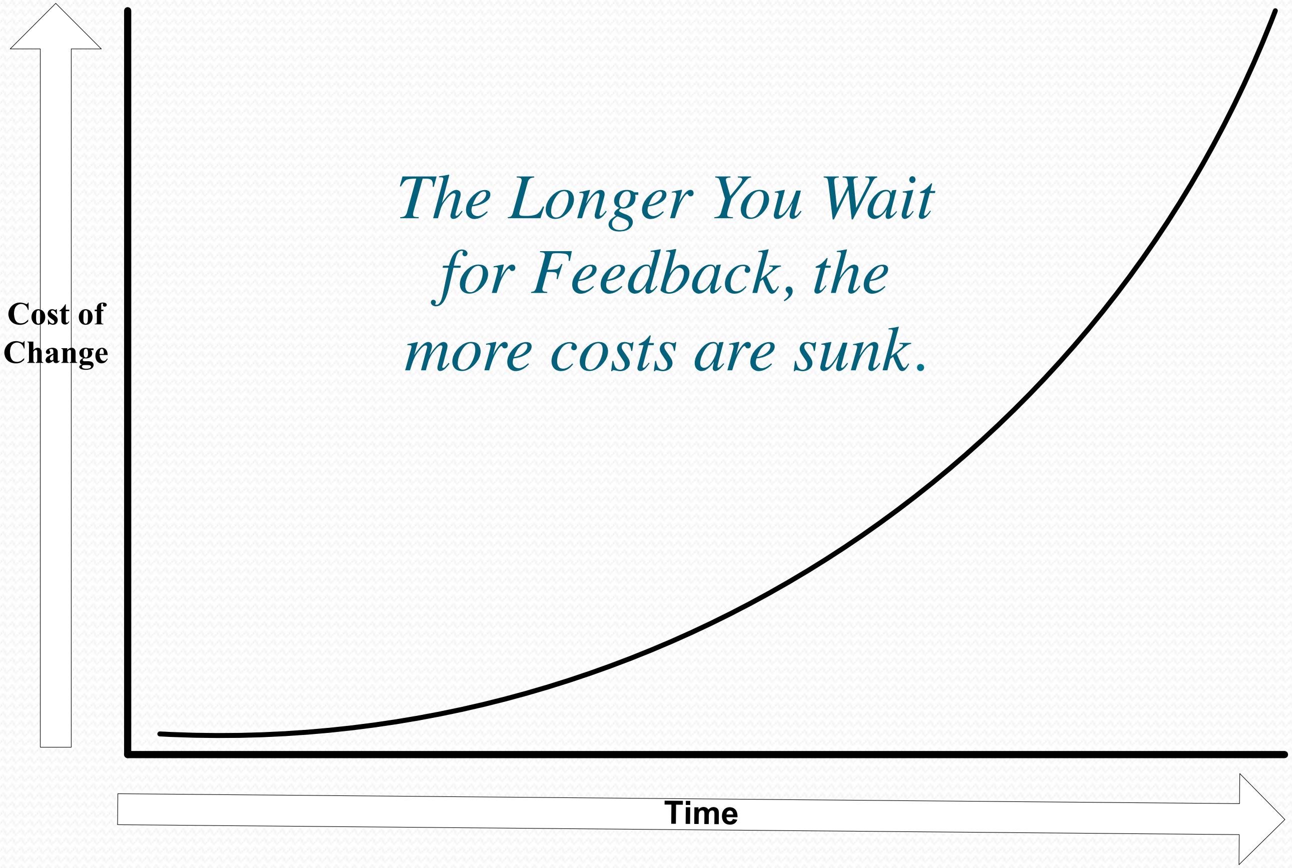
# Types of Software Prototypes

- **Throwaway prototype:**
  - Idea: creating a very rapid model (demonstration or marketing)
  - Early tool for envisioning software look and feel.
  - Could be paper: posters, screen sheets

- **Evolutionary prototype:**
  - Idea: Build the basic framework, flesh out
  - Lots of "stubs" that evolve in function
  - Pros: Build once, in slowly fleshing out stages
  - Cons: Less feel for big picture dynamics, inability to evolve technologies

- **Incremental:**
  - Idea: Build complete system…and then rebuild it…
  - …at different levels of quality/function/completeness
  - Pros: complete picture, ability to change decision for each increment
  - Cons: Waste of (re-)creating system several times

# Why? Serial development is costly!

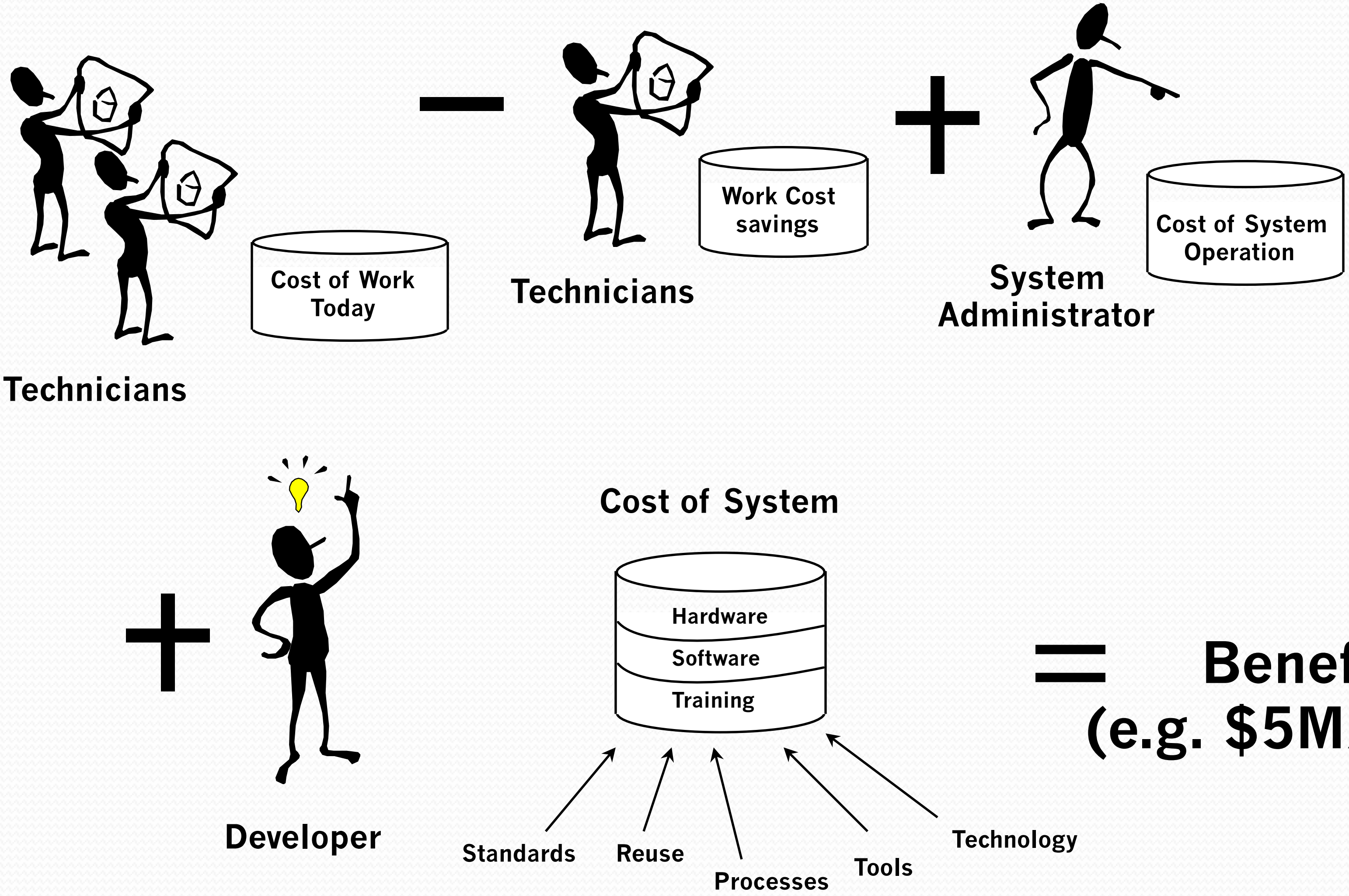


*The Longer You Wait for Feedback, the more costs are sunk.*

Prototyping of whole and/or key pieces allows for continuous re-targeting of design vision

# How does prototyping make sense?

## Cost/Benefit equation for Automation



Technicians

Cost of Work Today

–

Technicians

Work Cost savings

+

System Administrator

Cost of System Operation

+

Developer

Cost of System

Hardware

Software

Training

Standards

Reuse

Processes

Tools

Technology

=

Benefit

(e.g. $5M/year)

# Example Project A: *Order Reading and Analysis Software*

.

- **Business:**
  - Automated order processing
  - Interfaces with online submission, cust. Support, etc.
- **Size of Prototyping Effort:**
  - 12K lines of C Code (10% of final system module)
- **Purpose:**
  - Find a method for order reading and analysis, applicable to variable formats.
- **Duration and Staff of Prototype:**
  - Four people for eight months.

# Example A: *Prototyping Results*

Final requirements were based on prototype results.

→ more detailed insights

2. Alerted developers to the possibility of a having a tunable system.

   → improved solution.

3. Early evaluation of functional decomposition and performance showed bottlenecks in the dispatcher.

   → highlighted an area to guide implementation focus.

4. Eliminated the possibility of reusing code from another project.

   → Avoided later crash of project timeline

5. Prototype was thrown away due to decomposition and performance problems.

   → Prototype adequate to answer targeted questions…but no more.

# Project Example B: *Outside Plant Database System*

**Size of Effort:** 5% of 500K line of source code.

**Purpose:**

To evaluate database structures for an outside-of-plant database and to experiment with approaches to handling multiple future states of equipment usage.

**Duration:** Three people for 15 months.

# Example B: *Prototyping results*

1. A data base structure using hyper graph theory was invented.
   → develop/test a missing technological solution

2. An algorithm for handling both time-driven and event-driven assignments was invented
   → develop/test a missing technological solution

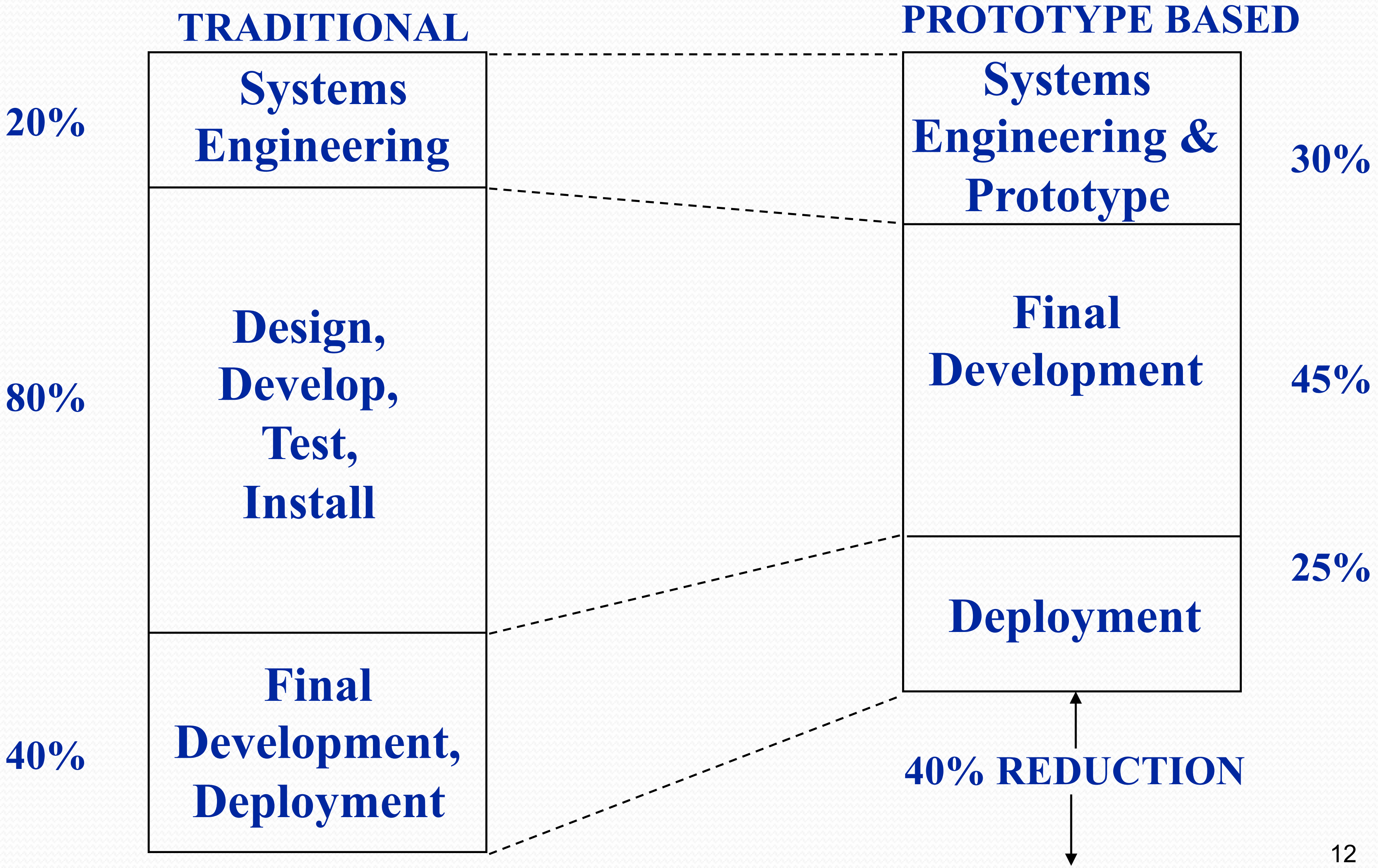
3. The prototype became the basis for the production code.
   → don’t always throw out the prototype! Re-implement algos in main (industrial strength) language.

4. UNIX flat files were used to model loop plant by way of a directed graph.
   → Quick and dirty way to “git er dun”. Cheap way that would never work for real system is fine to get prototype going fast.

5. The prototype showed database portability from Unix to the Mainframe.

   → Proves an important concept → refined project timeline

# Effects of good prototype utilization



TRADITIONAL

20% Systems Engineering

80% Design, Develop, Test, Install

40% Final Development, Deployment

PROTOTYPE BASED

Systems Engineering & Prototype 30%

Final Development 45%

Deployment 25%

40% REDUCTION

# So: Prototypes are vital

- **Benefits:**
  - Misunderstandings between software users and developers are exposed
  - Missing services may be detected and confusing services may be identified
  - A working system is available early in the process
  - The prototype may serve as a basis for deriving a system specification
  - The system can support user training and system testing

- **Keys to success:**
  - ✓ Plan to build a prototype
  - ✓ Document feedback from customer for the prototype.
  - ✓ Manage the feedback into the software product

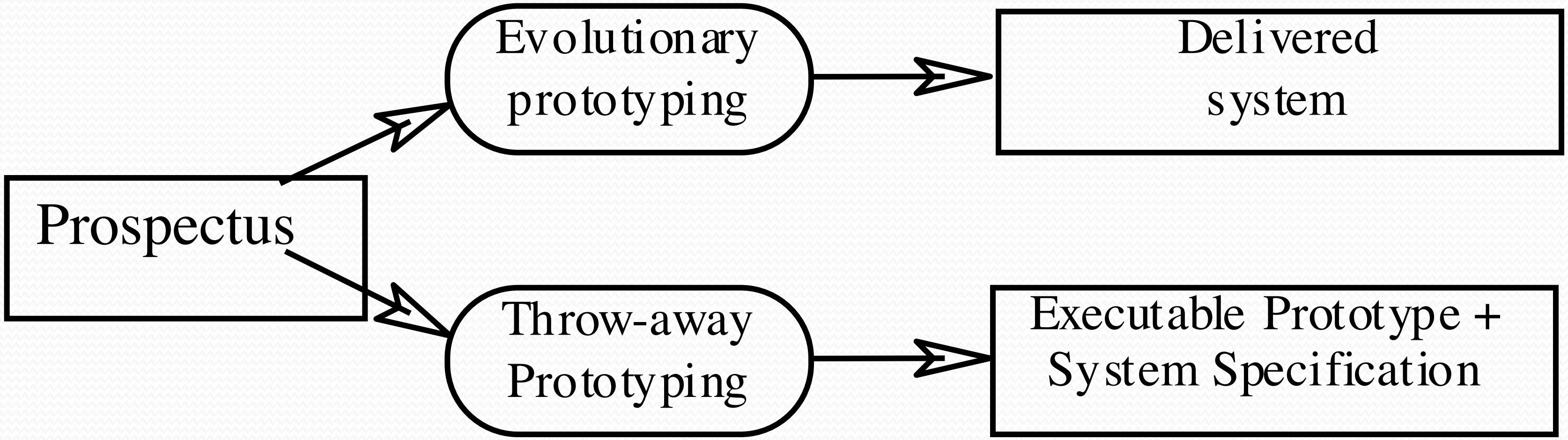
# Prototyping with Reuse

- **Idea: Don’t re-invent the wheel…until you need to**
  - Integrate existing systems to provide key functionality.
  - Adequate for prototype…but may need to re-implement later for efficiency, speed, or licensing reasons.

- **Application level development**
  - Entire application systems are integrated with the prototype so that their functionality can be shared
  - For example, if text preparation is required, a standard word processor software could be integrated

- **Component level development**
  - Piece together prototype from mutually independent components.
  - Individual components are integrated within a standard framework to implement the system
  - Framework can be a scripting language or an integration platform.

# Prototyping Approaches



Prospectus

Evolutionary prototyping

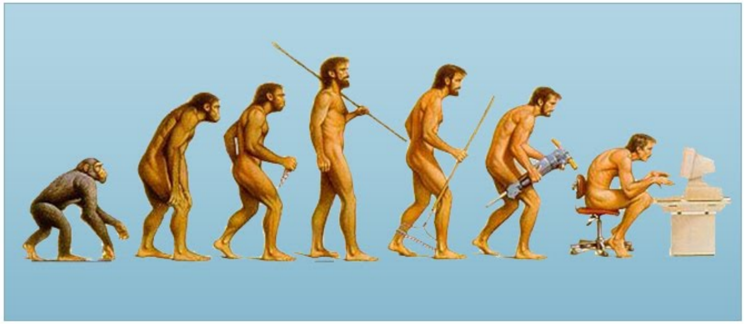
Delivered system

Throw-away Prototyping

Executable Prototype + System Specification

- **The objective of *evolutionary prototyping* is to deliver a working system to end-users**
  - **The development starts with those requirements which are best understood.**
- **The objective of *throw-away prototyping* is to validate or derive the system requirements**
  - **The prototyping process starts with those requirements which are poorly understood**

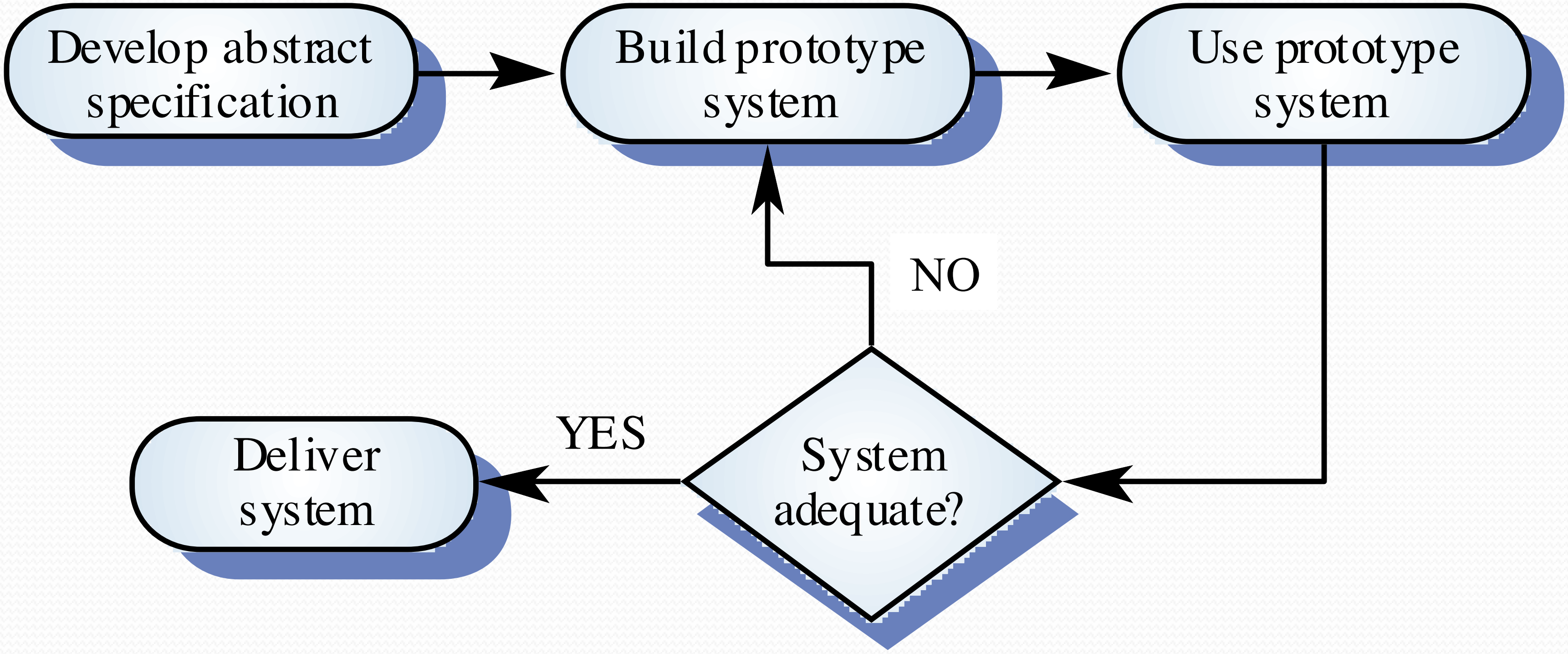
# Evolutionary Prototyping



- **Objective:**
  - deliver a working system to end-users
    - The development starts with those requirements which are best understood.
  - Used for systems where the requirements specification cannot be developed in advance
  - Based on techniques which allow rapid system iterations
  - Verification is impossible as there is no advance specification
  - Validation means demonstrating the adequacy of the system

http://base0.net/posts/big-bang-theory-intro-12-evolution-of-man/

# Evolutionary Prototyping Flow



# Evolutionary Prototyping Advantages

- **Accelerated (“continuous”) delivery of the system**
  - Rapid delivery and deployment of even a rudimentary system are sometimes more important than functionality or long-term software maintainability

- **User engagement with the system**
  - Users involved in development cycle.
  - Not only is the system more likely to meet user requirements, they are more likely to comm the system

# Evolutionary Prototyping Techniques

- Specification, design and implementation are intertwined
  - Agile methods shine here
- The system is developed as a series of increments that are delivered to the customer
  - Customer (or market) must be accepting of evolution
  - First iterations of actual system often very rough
- Techniques for rapid system development are used such as CASE tools and 4th Generation Languages (4GLs)
  - Visual programming, package-driven dev.
- User interfaces are usually developed using a GUI development toolkit → rapid prototyping, editing

# Evolutionary Prototyping Challenges

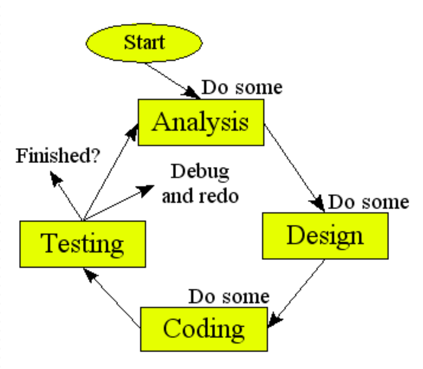
- **Management problems**
  - Existing management processes assume a waterfall model of development.
  - Specialist skills (agile, packages, etc.) are required which may not be available in all development teams

- **Maintenance problems**
  - Continual change tends to corrupt system structure so long-term maintenance is expensive

- **Contractual problems**
  - No specs?
  - How to specify final deliverable?

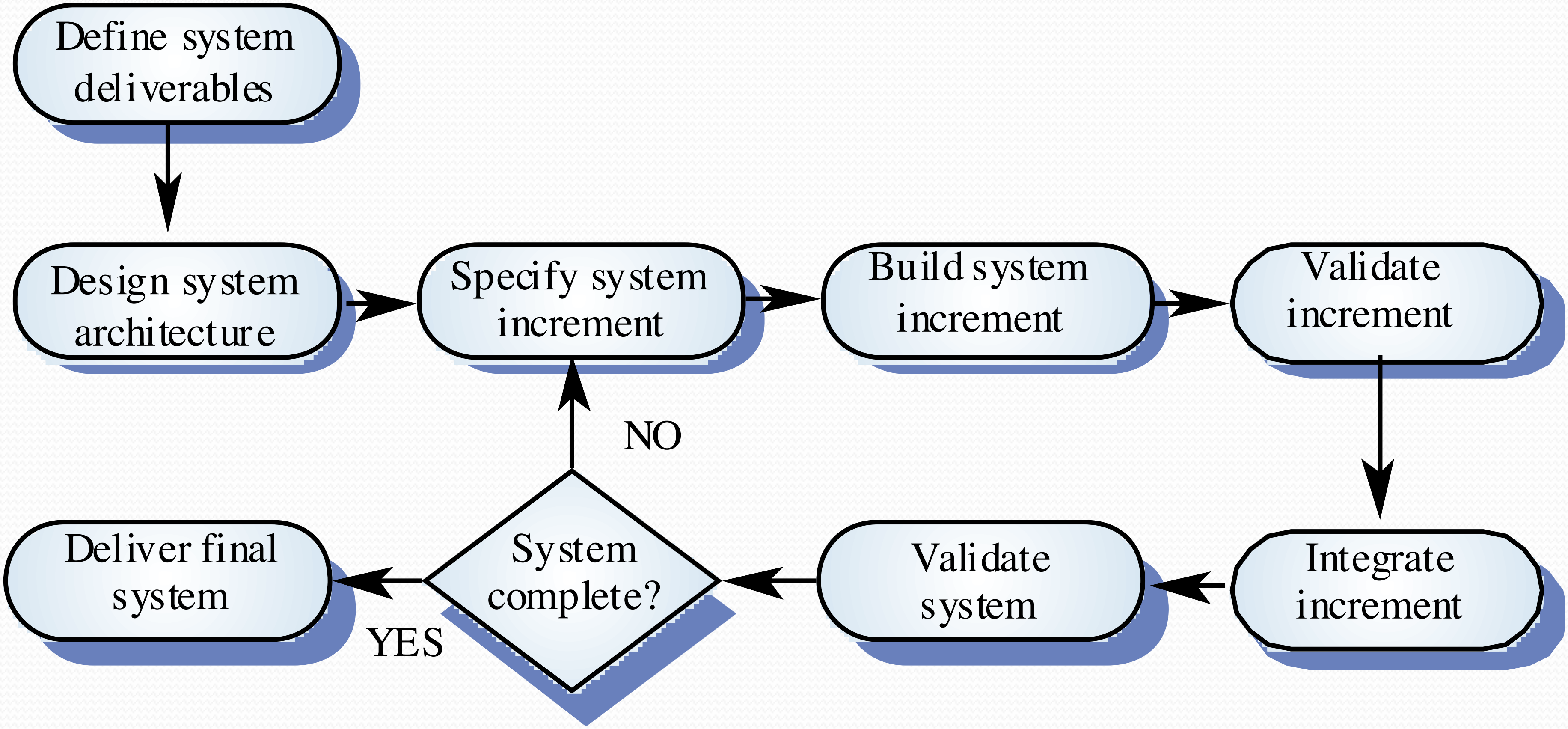
# Incremental Prototypes

- **System is developed and delivered in increments after establishing an overall architecture**



- **Requirements and specifications for each increment may be developed**
- **Users may experiment with delivered increments while others are being developed**
  - These serve as a form of prototype system
- **Intended to combine some of the advantages of prototyping**
  - More manageable process
  - Better system structure

# Incremental Development Flow



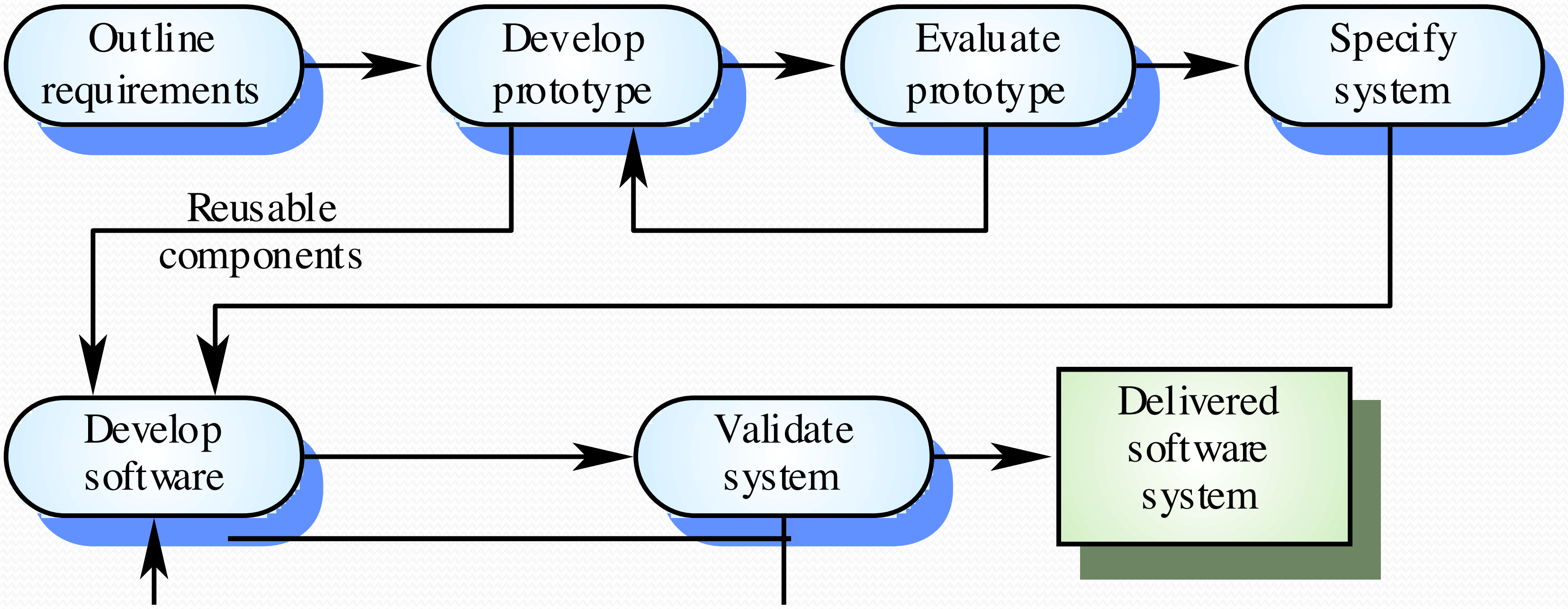
# Throw-away Prototypes

- Used to reduce requirements risk
  - When client is not sure of requirements/what they want
  - Not sure the vision they have would really work in real life

- The prototype is developed from an initial specification, delivered for experiment then discarded

- The throw-away prototype must NOT be considered as a final system
  - Some system characteristics may have been left out
  - There is no specification for long-term maintenance
  - The system will be poorly structured and difficult to maintain
  - Often written for speedy testing of concept…not efficiency
  - Often use rapid prototyping tools/languages.

# Throw-away Prototyping



# Part 2: Building Prototypes

- **Early prototyping focuses on *fast* exploration**
  - *Rapid Prototyping:* Typically small, focused *throwaway* prototypes
  - Emphasis: fast answers to specific question/risks

- **What to prototype?**
  - Principle 1: Focus on uncertainty and high risk issues
    - Performance of new algorithms or hardware
    - Critical choices of frameworks, libraries, subsystems (e.g. DB)
    - Critical user issues: Basic GUI metaphor, key actions/screens, visualizations
  - Principle 2: the 20/80 rule
    - Focus on the 20% of the functionality that will be used 80% of the time.
    - Develop complete set of users flows that cover all main interactions.
    - Requires very solid understanding of user domain and workflows!
  - Principle 3: Choose the right level of *fidelity* to target.
    - Focus on minimum function needed to answer the open spec. issue
    - Visual fidelity: rough paper sketches ← → High-fidelity graphics
    - Functional fidelity: (Semi-)Static elements ← → interactive responsiveness
    - Content fidelity: *lorem ipsum* ←→ actual domain content

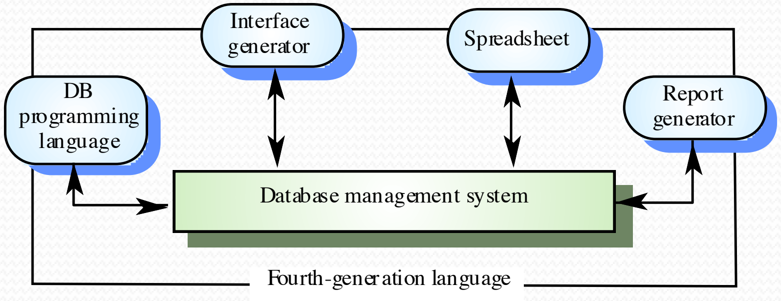
# Building Prototypes (cont.)

- **Different types of prototypes:**
  - <u>Database prototype:</u> Explore/develop appropriate database substrate to support project. Issues: speed, connectivity, robustness, etc.
  - <u>Performance prototype:</u> Explore/prove algorithms or hardware performance critical for system success
  - <u>User Interface prototype:</u> Explore key metaphor, layout and user flow questions in whole system, or in key modules/elements.

- **Need to develop effective prototyping plan:**
  - May have *several* focused prototypes to test key pieces
  - Each type of prototype has “best” languages/approaches
  - Develop complete plan to cover all critical/risky elements
  - Identify and prioritize “critical path” questions.
    - Things at the “design root”, that affect many other design decisions
    - Prioritize development of those. May influence need/nature of other prototypes

# Database Prototypes

- **Focus is on understanding data storage issues**
  - Complexity of data schema needed
  - Type of DB needed (relational, noSQL, flat file)
  - Capabilities: can chosen system support the DB interactions needed
    - Connectivity and programmatic access support (APIs)
    - Level of SQL implementation provided
    - Transaction rate. Roll-back/recovery options.
    - Special features: e.g. object-relational, distributed models
    - Cost/performance trade-offs.
- **Variety of Tools:**
  - ER-based DB generators
  - Simple and fast: MS Access
  - Actual systems plus their dev. Tools: MySQL, MongoDB
  - Full-on systems with strong DB integration: Ruby-on-Rails, some Webapp frameworks



# Performance prototypes

- **Goal: Explore mission critical performance challenges**
- **Type 1: Computational methods exploration**
  - Have a functional need, e.g., dynamic tree layout, large-scale search
  - Have idea for algorithmic solution. Or existing ones that *might* work.
  - Goal: Implement/port/adapt algorithm. Test on representative data.
  - Result: Is algorithm good fit for solving problem? Discover alternatives.
- **Type 2: Computational performance test**
  - Have functional need foreseeable in design
    - Search, sort, layout, rendering, etc.
    - Possibly on unusual (e.g. enormous) datasets
  - Have well-known existing software packages or hardware.
  - Goal: Test specific performance of available software/hardware on specific datasets/data volume anticipated in application.
  - Result: Clear performance data proving/rejecting solution.
- **Tools:**
  - Few shortcut tools. Must implement realistic test program in actual targeted implementation language
  - Key: narrowly implement *minimal* test harness!

# User interface prototyping

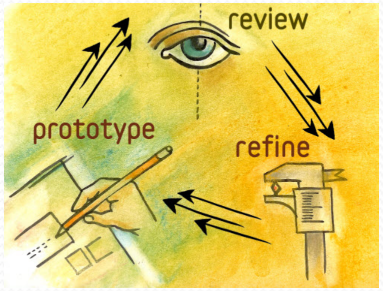
- **High impact: GUI development is often 80% of sys. cost!**

- **Goal: Explore fundamental GUI dynamics**
  - Impossible to textually specify the “look & feel” effectively. Must *see* it.
  - UI prototypes allow testing/refining dynamic aspect of user interaction.
  - Successful UI prototypes can be used to ground a system spec.

- **Wide variety of tools can be used:**
  - Static: pencil/paper, graphics editors (Omnigraffle)
  - Dynamic: Hypermedia (Director), Visual Basic, Widgets+scripting (TCK/TK).
  - Evolutionary tools: HLL languages with GUI builders (Xcode, Android IB)

- **Web interfaces:**
  - web site editors (Dreamweaver, etc.)
  - Pre-built JS packages (Bootstrap, etc.)

# Side Trip: Visual programming

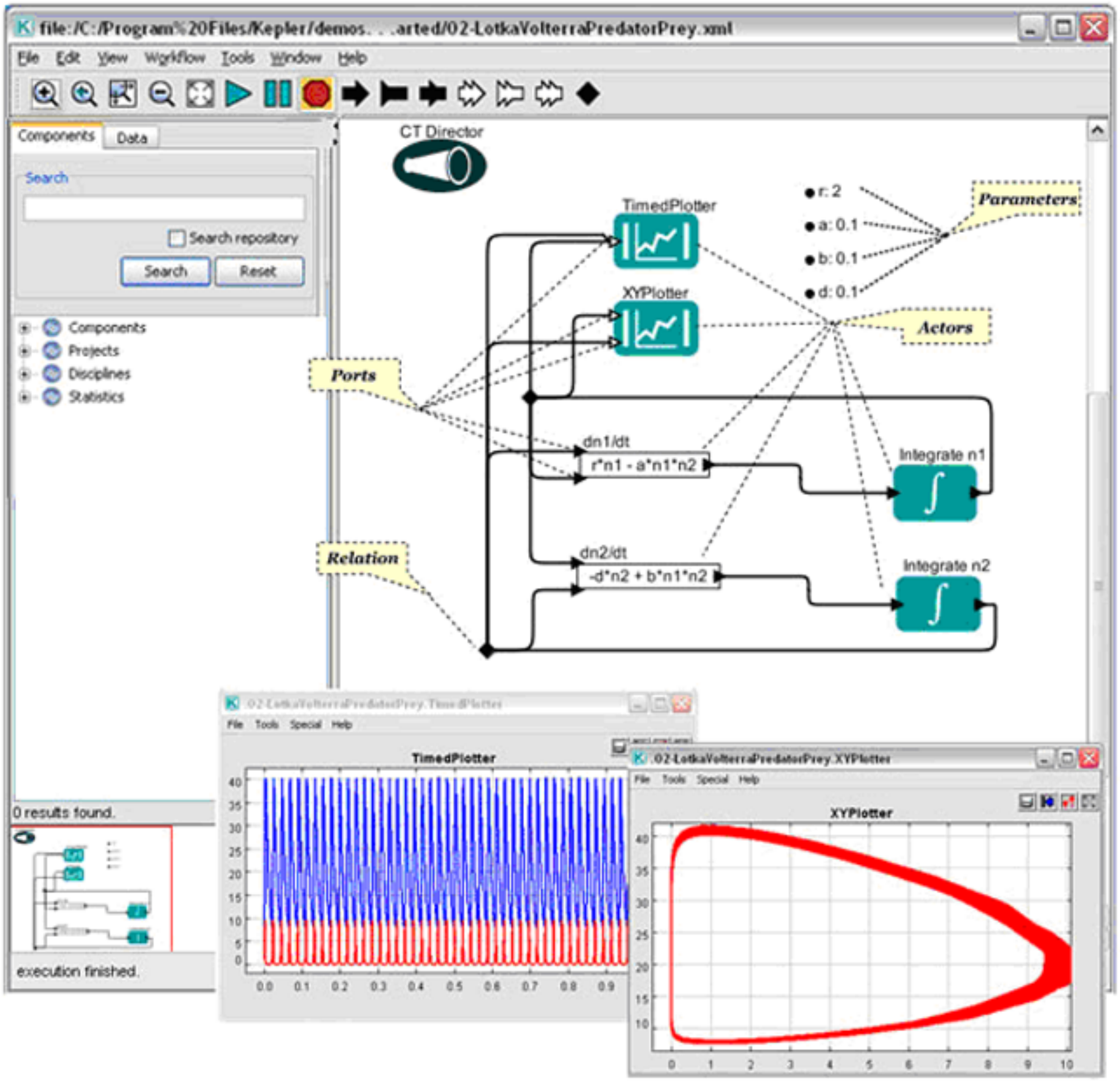


- **A broad metaphor in programming languages**
  - Idea: visually depict program function
  - How is works:
    1. drag-n-drop together pre-defined elements on a palette
    2. Configure and connect them together
    3. Actually *run* this visual representation
  - Fast edit-test cycle makes them ideal for rapid prototyping

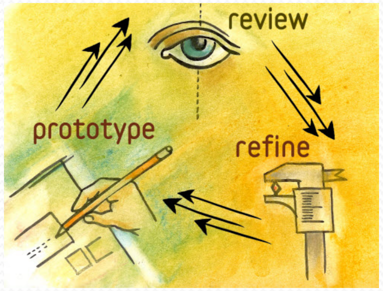
- **Type 1: Computational process modelling**
  - Draw out flow of computation
  - Nodes are pre-defined (configurable) processing tools
  - “Run” the visual network with input to produce result
  - Examples: Visual Basic, Labview, Alice, etc.

# Example: Computational Process Modeling

- **Kepler Project**
- Modeling scientific data processing workflows
- Could be used to model any data processing and manipulation
- Includes large library of configurable processing nodes.
- Includes many data visualization and graphing tools.
- Widely-used!



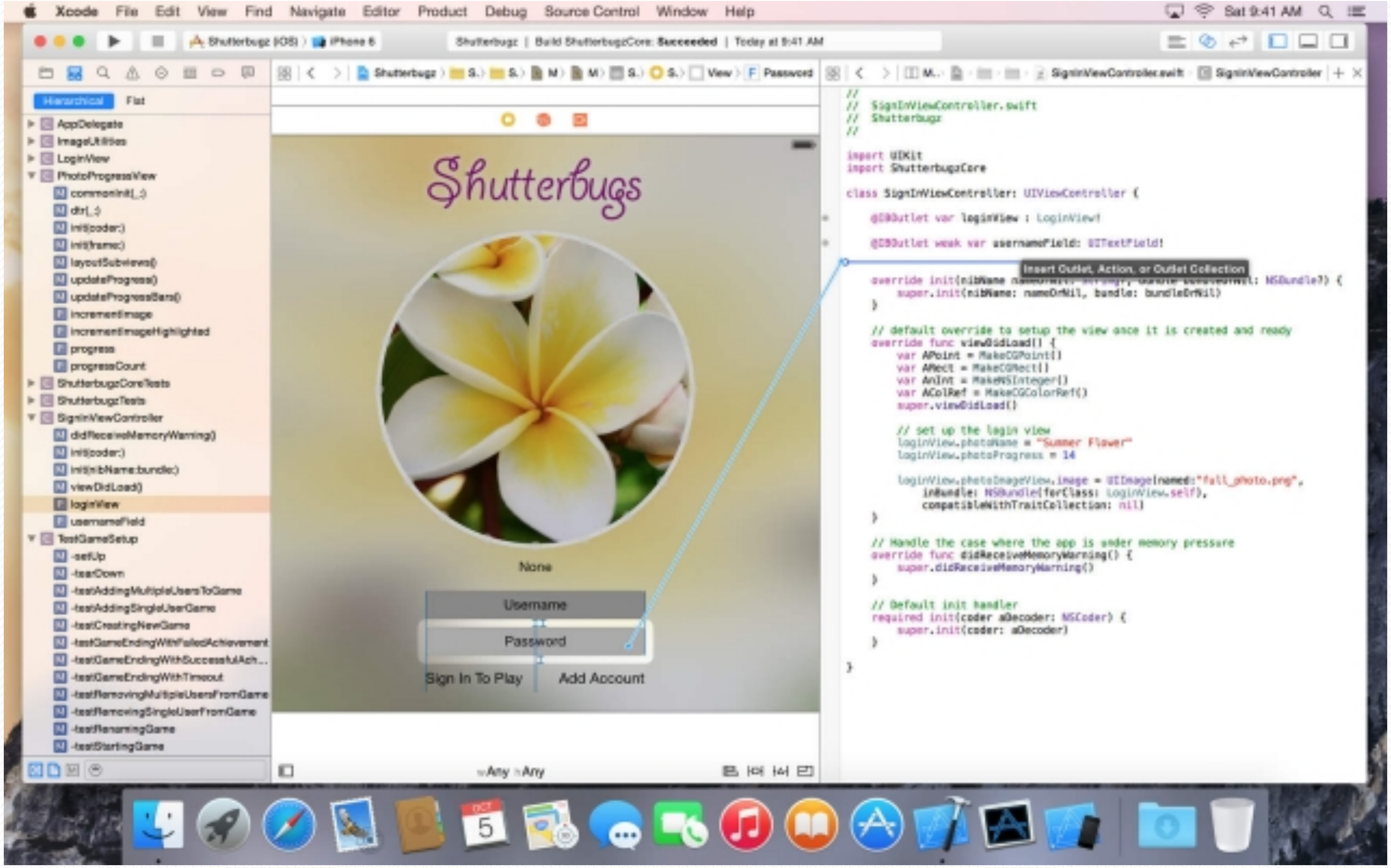
# Visual programming



- **Type 2: User Interface modelling**
  - WSYWYG GUI design: rapidly lay out UI
  - Large selection of GUI elements or “widgets”
    - Physical: Windows, buttons, menus, menu items
    - Logical: Layout managers, active modules (e.g. login dialog)
  - Laid out on the actual viewport/real estate of target UI
    - Direct representation: Large screen, small screen, mobile device
  - Dynamic behavior to connection to events and handlers
    - May have “default” events and handlers built in.
  - Works well for “standard” GUIs
    - Standards GUI elements and interaction techniques
    - No support for custom graphics or interactive behaviors
  - Examples: Xcode GUI builder, Android Dev. Kit, some website/ webapp toolkits.

# Example: Xcode Interface Builder

- WSYWYG layout of elements
- Connect events to handler code
- Great to prototype…
- …then flesh out for full application functionality
- Fully integrated in Apple software IDE: remains active throughout coding phase.


Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help
Sat 9:41 AM
Shutterbugz | Build ShutterbugzCore: Succeeded | Today at 9:41 AM
Hierarchical Flat
SignInViewController.swift
SignInViewController
Shutterbugs
None
Username
Password
Sign In To Play
Add Account
Insert Outlet, Action, or Outlet Collection
wAny hAny

```
//
//  SignInViewController.swift
//  Shutterbugz
//

import UIKit
import ShutterbugzCore

class SignInViewController: UIViewController {

    @IBOutlet var loginView : LoginView!

    @IBOutlet weak var usernameField: UITextField!

    override init(nibName nameOrNil: String?, bundle bundleOrNil: NSBundle?) {
        super.init(nibName: nameOrNil, bundle: bundleOrNil)
    }

    // default override to setup the view once it is created and ready
    override func viewDidLoad() {
        var APoint = MakeCGPoint()
        var ARect = MakeCGRect()
        var AnInt = MakeNSInteger()
        var AColRef = MakeCGColorRef()
        super.viewDidLoad()

        // set up the login view
        loginView.photoName = "Summer Flower"
        loginView.photoProgress = 14

        loginView.photoImageView.image = UIImage(named:"fall_photo.png",
            inBundle: NSBundle(forClass: LoginView.self),
            compatibleWithTraitCollection: nil)
    }

    // Handle the case where the app is under memory pressure
    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }

    // Default init handler
    required init(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }

}
```

# Challenges with Visual Programming

- **Weak system architecture**
  - Grouped by graphics, not logic
  - (re)factoring?
  - Complex dependencies
    - Can have large number of connections of elements
    - Every connection creates dependency
    - Debugging and Maintainability challenges
- **Can be difficult to coordinate team development**
  - Often poor modularity
  - Depencies + multiple programmers = confusion

"You can't just punch in 'let there be light' without writing the code underlying the user interface functions."

© 1997 J. P. Rini from The Cartoon Bank. All rights reserved.

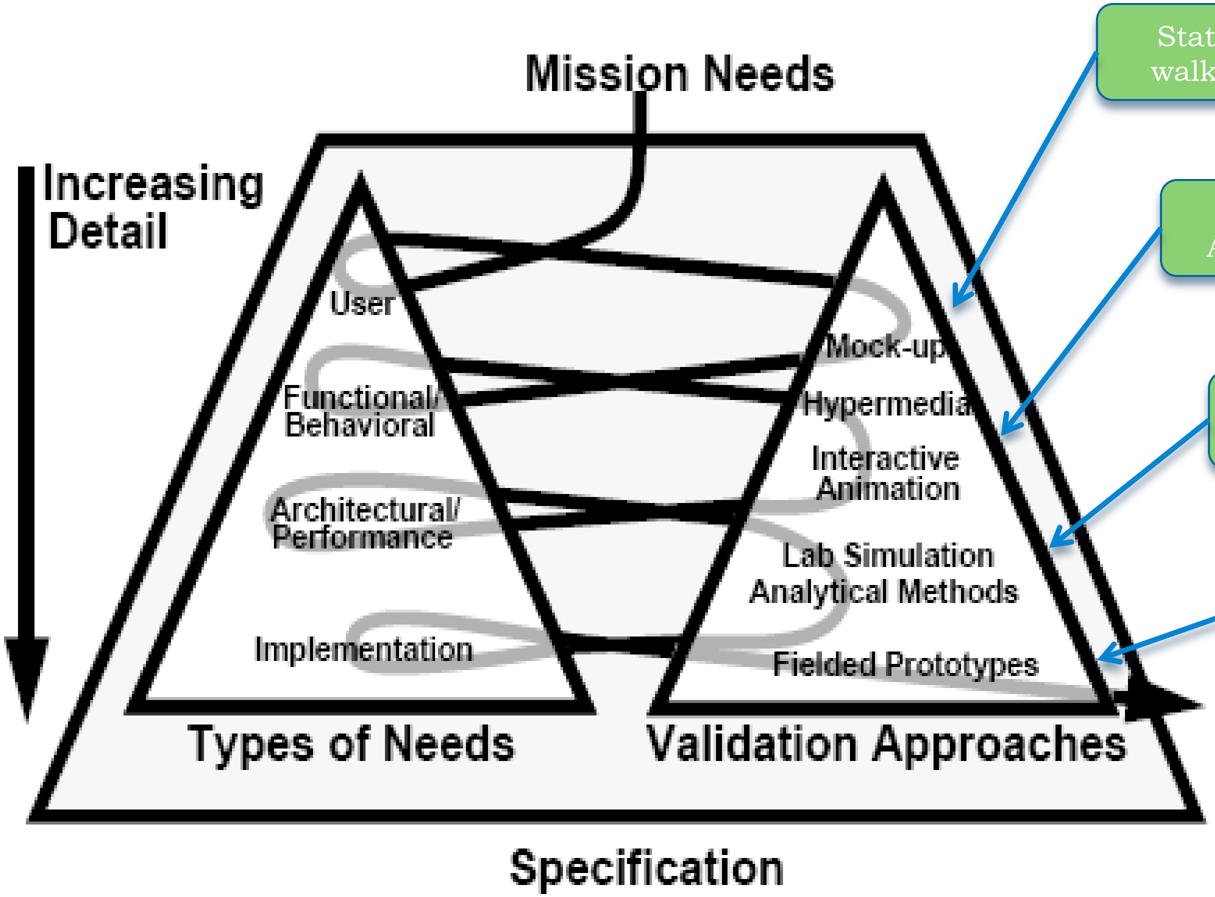
http://www.pcmag.com/encyclopedia/term/53558/user-interface

# Summary: Prototyping The Right Mindset

- **Commitment to the concept**
  - Prototyping as critical for system parts that cannot be effectively pre-specified
- **Focus:**
  - Narrowly on mission-critical risk/issues
  - *Partial* functionality. What is needed only.
- **Tools:**
  - Choose the right tools for fastest create/review cycle
  - Hypermedia, scripting, visual programming, existing packages
  - For web apps: sites that provide infrastructure (DBs, frameworks)
- **Involve End-users**
  - Users must be involved in prototype evaluation
  - For GUIs…but also for performance tests and other “content”

# Summary: Prototyping

## Multiple prototypes to progressively develop spec.



Static screen walkthroughs

Visual Basic, Adobe Director

Testbeds, performance tests

Rough implementations of targeted features

# Software Prototyping: *Review*

- **Types: Throwaway, Evolutionary, Incremental**
  - Each has advantages and disadvantages
  - Choice depends on:
    - Timeline: how fast does user need product
    - Prototyping goals: a small part, the whole thing?
    - Users: how tolerant to partial, evolving functionality
    - Others:  need to consider context carefully

- **Prototyping Tools:**
  - Paper/pencil, drawing packages,
  - hypermedia, scripting languages,
  - WSYWYG GUI generators.
  - Libraries and development frameworks
  - Simulation technologies

THE PROTOTYPE IS DONE. COME TAKE A LOOK AT THE USER INTERFACE.
IT WORKS GREAT, BUT MAKE SURE THIS THING IS TOTALLY IDIOT—PROOF.
AGAIN?
www.dilbert.com
scottadams@aol.com
7-22-09 © 2009 Scott Adams, Inc./Dist. by UFS, Inc.