Skills for the Agile Designer

Rebecca Wirfs-Brock
rebecca@wirfs-brock.com
www.wirfs-brock.com

What makes a designer agile?

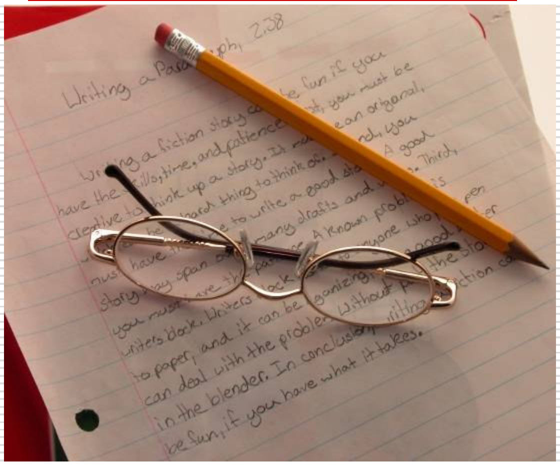
- Core values:
  - Simplicity
  - Communication
  - Learning
  - Teamwork
  - Trust
  - Satisfying stakeholder needs
- An attitude

How can you become more agile?

- Learn fundamental strategies for producing acceptable solutions
- Be curious. Learn from mistakes and successes
- Practice different ways of seeing the nature of problems and solutions
- Communicate

Tools for Seeing

A Designer’s Story: A tool for seeing what’s important



- Designer’s story—a quickly written paragraph or two description of important ideas, what you know, and what you need to discover

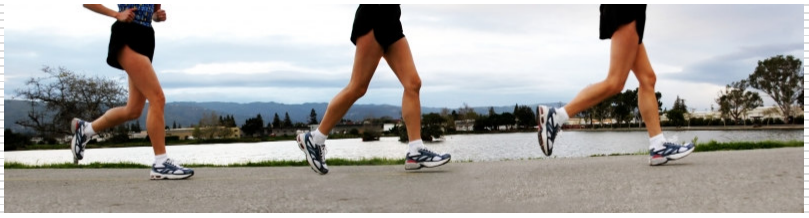
Elements of a story…

- What is your design supposed to do?
- Is there something similar you can draw upon or emulate?
- What will make it a success?
- What are the most challenging parts?

Why tell a designer's story?

- To put your spin on what's important
- Describing the problem helps you own it
- Sharing them builds understanding and a common vision
- Metaphors are hard to come by…identifying themes and key responsibilities from designer stories is one alternative

Write a Designer's Story

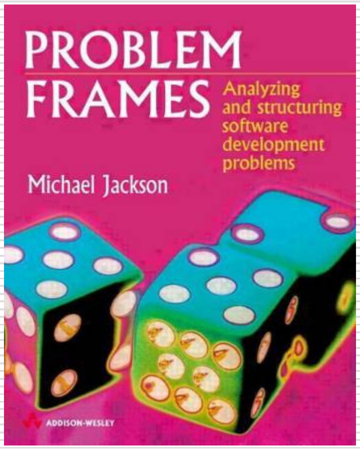


- Need to take data from various sensors and store as "normalized" measurements
- Need to set up and monitor sensors that are either polled or programmed to respond on a specified time interval
- Need to analyze data and make predictions

Problem Frames: A tool for seeing typical patterns of software tasks

frame—a structure that gives shape or support

A Problem Frame…



"… defines the shape of a problem by capturing the **characteristics and interconnections of the parts of the world it is concerned with**, and the **concerns** and **difficulties** that are likely to arise. So problem frames help you to focus on the problem, instead of drifting into inventing a solution."

—Michael Jackson

Framing strategy: divide and conquer

- Decompose problems
- Focus on the requirements and the concerns of each subproblem

Tactics for decomposing problems

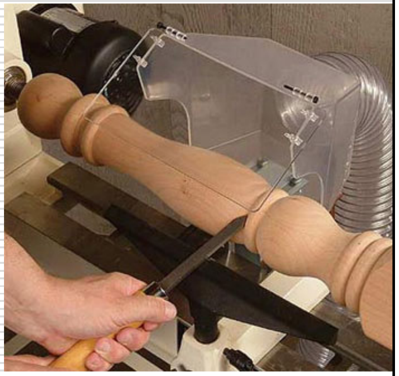
- Identify the core problem
  - Sending and receiving mail
- Look for ancillary problems
  - Constructing mail
  - Managing mail folders
  - Sorting mail
  - Reading mail
  - Maintaining address lists
- Examine problem concerns for more

Five basic problem frames

- **Workpiece-** a tool that allows users to create and manipulate structures (Email editor)
- **Transformation-** converting input source to some output according to certain rules (MIME Decoding)
- **Information-** information is needed to be derived from something's observed state and/or behavior (Determining a "junk mail" rating)
- **Commanded behavior-** controlling the behavior of some "thing" according to operator commands (Sending an email message)
- **Required behavior-** controlling the behavior of some "thing"
  (Sorting incoming mail according to pre-defined filters)

Stylized Workpiece Questions

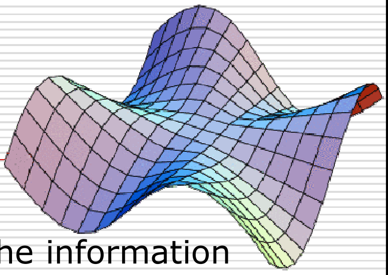
- Will it take different forms?
- Does it have an interesting lifecycle (or is it something that is changed and then treated as "static" after each change)?
- Is it passed around between various users? Is there a workflow associated with it?
- Should it be published or printed?



Stylized Transformation Questions

- What do you start with?
- How will it be changed?
- Is the transformation complex?
- Will it always work? What should happen when you encounter errors in the input?
- Is the transformation "lossy" or reversible?
- What speed, space, or time tradeoffs are there?

Stylized Information Display Questions



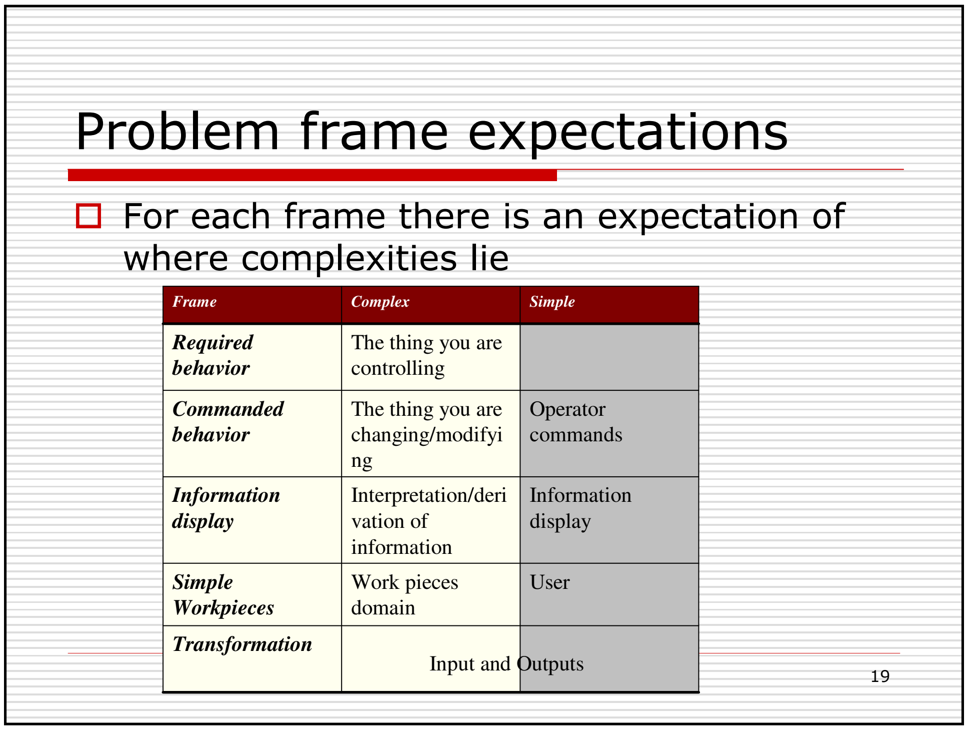
- How precise does the information need to be? Is the information "fuzzy"?
- How much computation does your software have to do to come to an observation?
- Is the user only interested in current information? Or is historical information important?
- Are there questions that the user may want to ask about the information? What are they? How easy are they to accurately answer?

Stylized Commanded Behavior Questions

- What does the use need to know to "command" the system to do things?
- Do certain commands need to be inhibited? Do they always make sense?
- Is there a lag between issuing and performing the command?
- What happens when a command fails?
- Should certain commands be ignored?
- Do commands need to be reversible? logged? monitored or tracked?

Stylized Required Behavior Questions

- What external state must be controlled?
- How does your software find out whether its actions have had the intended effect?
- What should happen when things get "out of synch"?
- How does your software decide what actions to initiate?
- Is there an action sequence?
- Are there complex interactions with your software and the thing under its control?

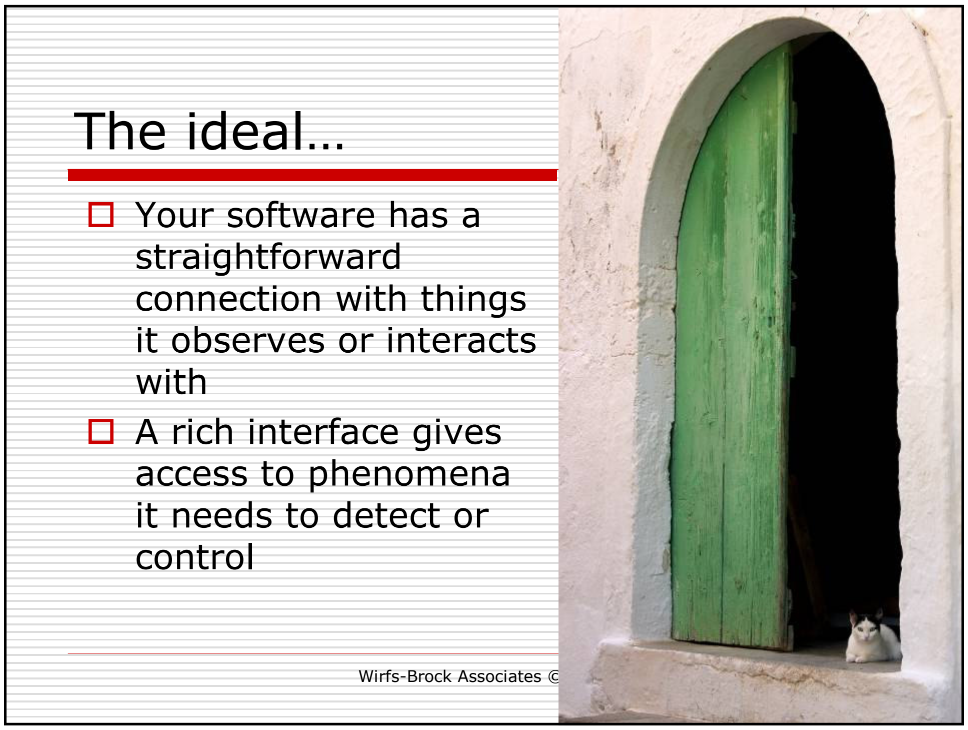


Problem frame expectations

- For each frame there is an expectation of where complexities lie

Frame	*Complex*	*Simple*
Required behavior	The thing you are controlling	
Commanded behavior	The thing you are changing/modifying	Operator commands
Information display	Interpretation/derivation of information	Information display
Simple Workpieces	Work pieces domain	User
Transformation	Input and	Outputs

19



The ideal…

- Your software has a straightforward connection with things it observes or interacts with
- A rich interface gives access to phenomena it needs to detect or control

Wirfs-Brock Associates ©

Reality…

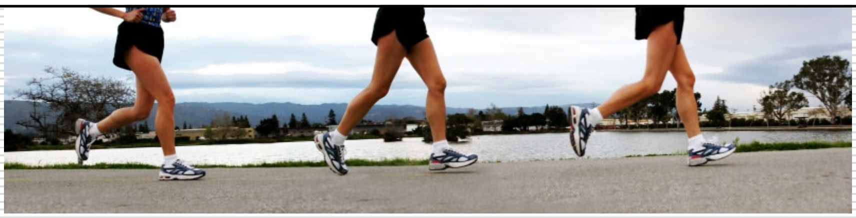
- An "intermediary" often lies between
- This connection can be quirky
  - Design your software to react in the face of potential time-delays, conflicting states between the "connected" thing you are controlling or interacting with

What framining in an agile world?

- Jackson advocates fully understanding problems before starting design
- Agile developers expect to incrementally discover requirements. Framing
  - …can focus design spikes
  - …can help you write tests
  - …can help you estimate stories
  - …can lead to deeper understanding of user stories and dependencies

Where problem frames don't fit

- Framing doesn't help describe or understand:
  - Mathematical computations or algorithms
  - Graphics user interfaces
  - Compiler design
  - …



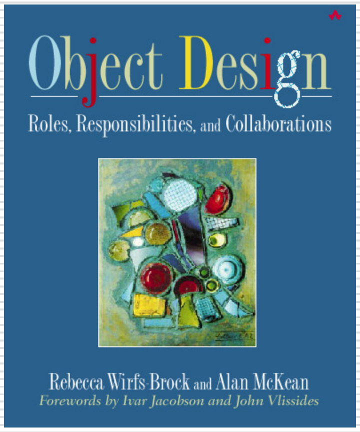
Frame the First Release

- Release 1 will only process data from programmable sensors (not polled sensors yet). We need to program them to report on a specified interval, and then receive and process their measurement data
- Identify the problem frames in this first release
- List clarifying questions for one frame

Thinking in concepts…

vegetarian

Role Stereotypes


Object Design
Roles, Responsibilities, and Collaborations
Rebecca Wirfs-Brock and Alan McKean
Forewords by Ivar Jacobson and John Vlissides

"A well-defined object supports a clearly defined role. We use purposeful oversimplifications, or **role stereotypes,** to help focus an object's responsibilities…Once we assign and characterize an object's role, its attendant responsibilities will follow."

—Rebecca Wirfs-Brock & Alan McKean

Role Stereotypes: A tool for seeing and shaping object behaviors

- stereotype—A conventional, formulaic, and oversimplified conception, opinion, or image

From Responsibility-Driven Design: Object Role Stereotypes

- Information holder

knows and provides information

Measurement

- Structurer - maintains relationships between objects and information about those relationships

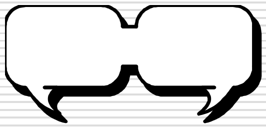
SensorRepository, PollingSchedule

Object Role Stereotypes

- Coordinator – mechanically reacts to events **SensorPoller**

- Controller - makes decisions and closely directs others' actions **DataCollector**

Object Role Stereotypes



- Interfacer - transforms information and requests between distinct parts of a system **Sensor**

- Service provider - performs work on demand **ConfidenceRater**

Three Uses for Object Role Stereotypes

- Early, stereotypes help you think about the different objects you'll need
- Blend stereotypes to make objects more responsible and intelligent
  - information holders that compute
  - service providers that maintain information
  - interfacers that transform information and hide low-level details
- Study a design to learn what types of roles predominate and how they interact

CRC Cards: An informal tool

Candidate, **R**esponsibilities, **C**ollaborators

Sensor

Purpose: Represents what the Arbor 2000 system knows about devices that reports data that is physically sensed from the environment. Sensors can report light intensity, temperature, wind speed and direction, rainfall and other physical readings. Some kinds of sensors can sense multiple physical characteristics and are capable of reporting readings at different intervals (such as every minute, hourly, weekly, monthly) or based on a significant event (temperature rising x degrees in a period of time, x amount of rainfall, etc.).

Stereotypes: Service Provider, Interfacer

CRC Cards: An informal tool

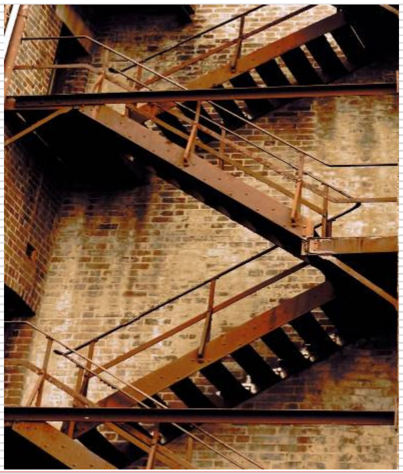
Candidate, **R**esponsibilities, **C**ollaborators

Sensor	
knows physical characteristics it can detect	Parser
knows reporting interval	Measurement
maintains configuration parameters	
knows sensor make and model	
knows location	
knows if activated	
creates measurements from reports	

Collaborators

Responsibilities

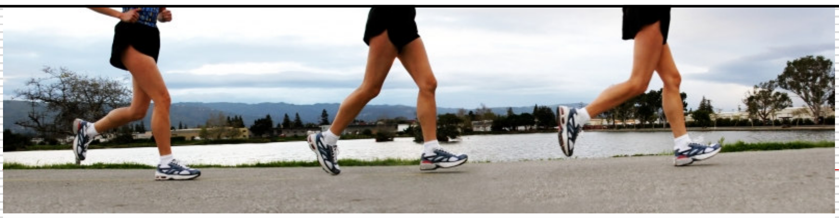
Seeing abstractions



- We can see objects and behavior at different levels:
  - At the **conceptual** level- a set of responsibilities
  - At the **specification** level- set of methods that can be invoked
  - At the **implementation** level- code and data

n-tier web applications

Layer	Functionality	Role	Technique
Client	User Interface	**Interfacer**	HTML, JavaScript
Presentation	Page Layout	**Interfacer**	JSP
Control	Command	**Coordinator**	Servlet
Business Logic	Business Delegate	**Controller**	POJO, Session EJB
Data Access	Domain Model	**Information Holder, Structurer**	JavaBean, Entity EJB
Resources	Database, Enterprise Services	**Service Provider**	RDBMS, Queues, Enterprise Service Bus

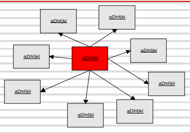


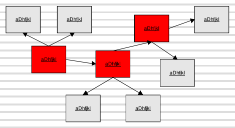
Identify Candidates

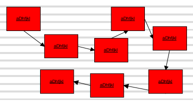
- In 10 minutes, come up with a list candidates and their stereotypes:
  - What work needs to be done? (Controllers, Coordinators, Service Providers)
  - What information does the software need to track and/or produce? (Information holders)
  - What needs to be structured and managed? (Structurers)
  - How does it connect to other systems and external devices? (Interfacers)

Tools for Shaping

Control centers and collaboration styles: Tools for shaping solutions







control center—a place where objects charged with controlling and coordinating reside

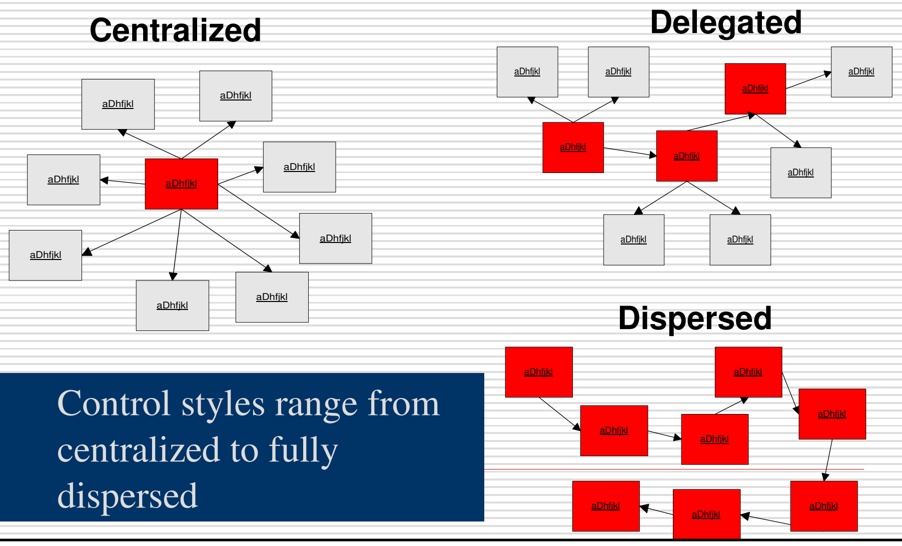
Control Centers

- Deciding on and developing a consistent control style is one of the most important design decisions. There are many control centers in your design, each may have a different style
  - Handling web interactions
  - Managing complex software processes
  - Designing how objects work together within a subsystem
  - Controlling external devices or external applications

Control Design

- Involves decisions about
  - how to control and coordinate tasks,
  - where to place responsibilities for making domain-specific decisions (rules), and
  - how to manage unusual conditions (the design of exception detection and recovery)
- Goal: develop patterns for distributing the flow of control and sequencing of actions among collaborating objects. Make similar parts of your system consistent

Collaboration Styles

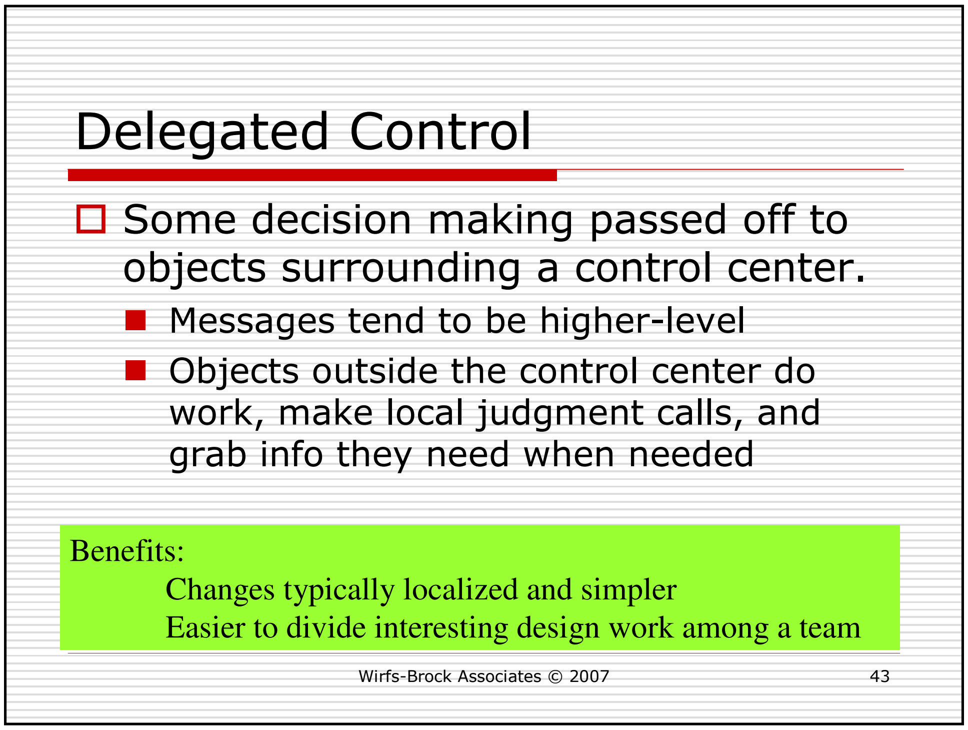


Control styles range from centralized to fully dispersed

Centralized Control

- Generally, one object (the controller) makes most important decisions. Tendencies to avoid:
  - Overly complex control logic
  - Changes rippling among controlling and controlled objects
  - Objects coupled indirectly through controller actions

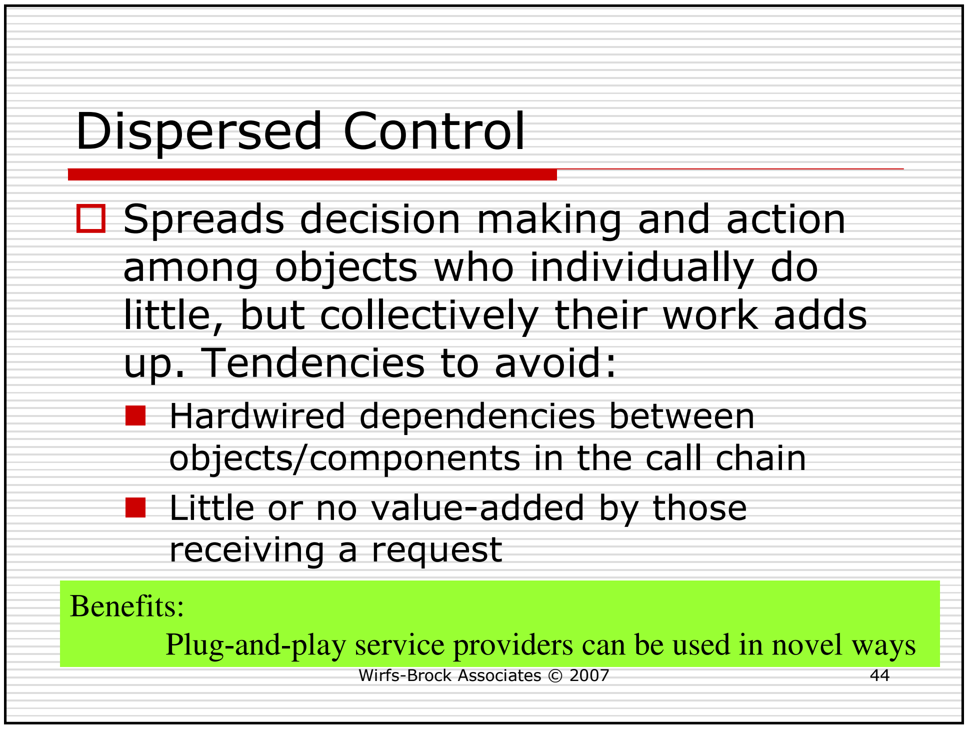
Benefits:
Easy to see what's going on. Easy to test.



Delegated Control

- Some decision making passed off to objects surrounding a control center.
  - Messages tend to be higher-level
  - Objects outside the control center do work, make local judgment calls, and grab info they need when needed

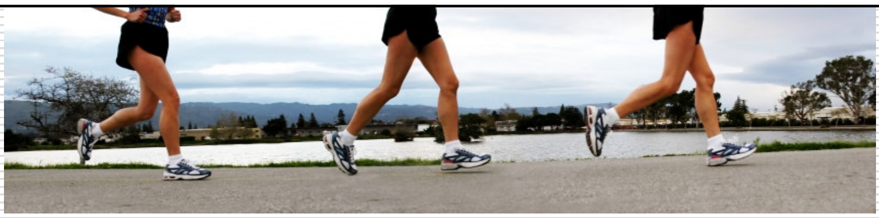
Benefits:
    Changes typically localized and simpler
    Easier to divide interesting design work among a team



Dispersed Control

- Spreads decision making and action among objects who individually do little, but collectively their work adds up. Tendencies to avoid:
  - Hardwired dependencies between objects/components in the call chain
  - Little or no value-added by those receiving a request

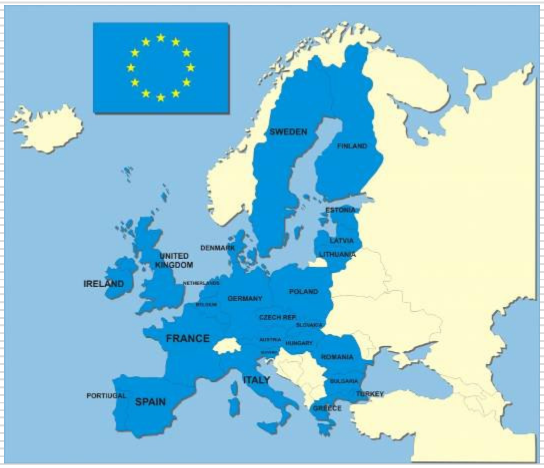
Benefits:
    Plug-and-play service providers can be used in novel ways



Explore Control Style Elements

- You need to keep track of polling intervals and what sensors need to be polled
- A sensor's polling interval can be changed by the user
- A system has fewer than 100 sensors
- Discuss/sketch your ideas for handling polling

Trust Regions: A tool for seeing where "defensive" behavior is or isn't needed



trust region—an area where trusted collaborations occur

Definition: Collaborate

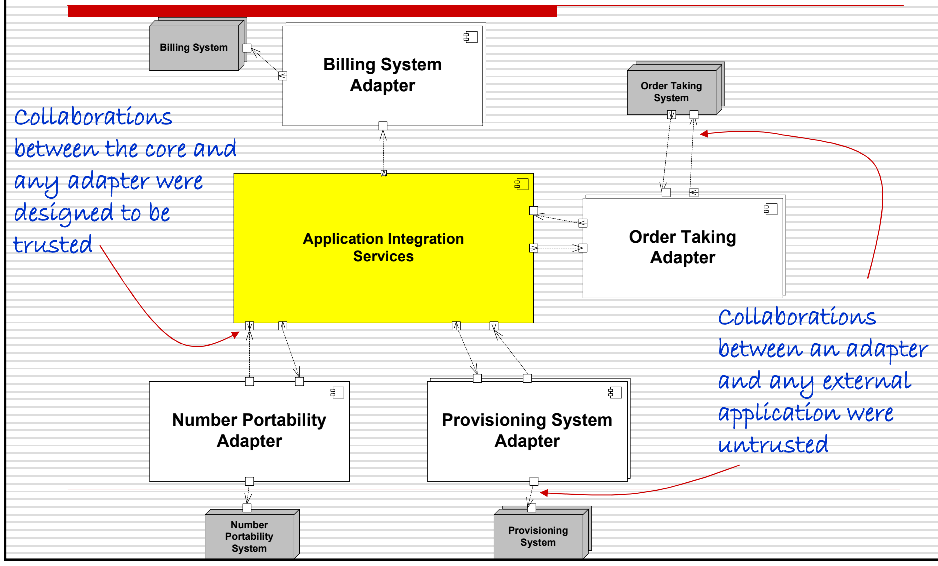
1. To work together, especially in a joint intellectual effort

2. To cooperate treasonably, as with an enemy occupation force

Implications of trust

- Components and objects at the "borders" may take on extra responsibilities
- Within a trust region, collaborations can be more collegial
  - Check once, then proceed…
  - Code deep inside a trust region need not check for well-formed or timely requests

Trust In A Telco Integration Application

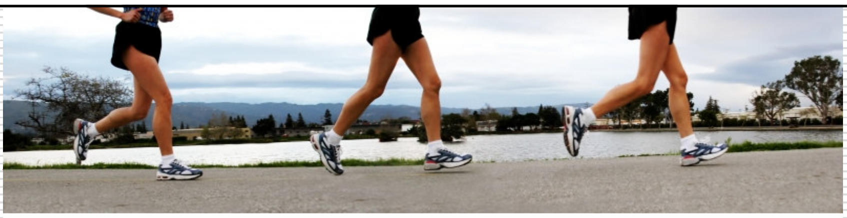

Billing System
Billing System Adapter
Order Taking System
Application Integration Services
Order Taking Adapter
Number Portability Adapter
Provisioning System Adapter
Number Portability System
Provisioning System

Collaborations between the core and any adapter were designed to be trusted

Collaborations between an adapter and any external application were untrusted

Collaboration Cases To Consider

- Collaborations between objects or components…
  - that interface to the user and the rest of the system
  - in different layers or subsystems
  - inside your system that interface to external systems
  - you design and those designed by someone else



Identify Trust Regions

- Where are the trust boundaries in the system?
  - Our system receives data from physical sensors that are self-reporting or polled
  - Measurement data is stored and analyzed to make predictions and analyze weather trends
  - Vendor supplied or open source plug-ins can provide additional tools for data analysis and visualization

Tools for Design Collaboration

Design problem types: A tool for balancing priorities

Each type of design problem category warrants a different approach and has a different rhythm to its solution

Photo courtesy Drum Journey www.drumjourney.com

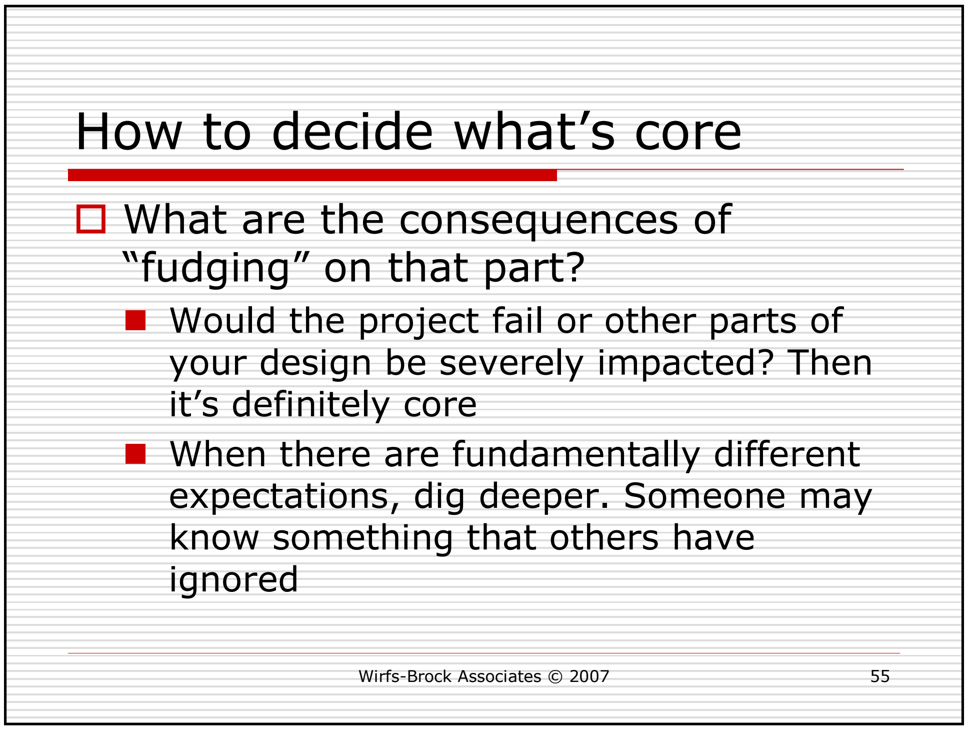
53

Design tasks aren't alike

- **Core design problems** include those fundamental aspects that are essential to your software's success
- **Revealing design problems** when pursued, lead to a fundamentally new, deeper understanding
- **The rest,** while not trivial requires far less creativity or inspiration

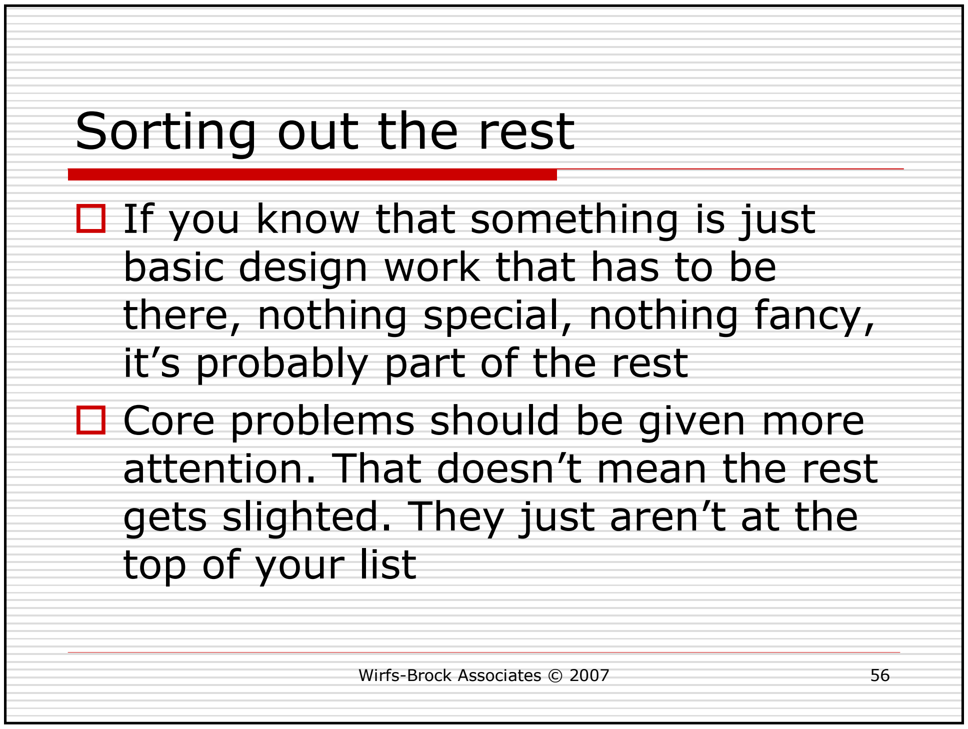
Wirfs-Brock Associates © 2007

54



How to decide what’s core

- What are the consequences of “fudging” on that part?
  - Would the project fail or other parts of your design be severely impacted? Then it’s definitely core
  - When there are fundamentally different expectations, dig deeper. Someone may know something that others have ignored



Sorting out the rest

- If you know that something is just basic design work that has to be there, nothing special, nothing fancy, it’s probably part of the rest
- Core problems should be given more attention. That doesn’t mean the rest gets slighted. They just aren’t at the top of your list

Revealing design problems

- What distinguishes revealing problems is their degree of difficulty and the element of surprise, discovery and invention

Wirfs-Brock Associates © 2007 57

Revealing design problems are always hard…

- coming up with a solution is difficult—even though it may be straightforward
- they may not have a simple, elegant solution
- they may not be solvable in a general fashion—each maddening detail may have to be tamed, one at a time
- they may require you to stretch your thinking and invent things

Wirfs-Brock Associates © 2007 58

Some problems are **really** hard

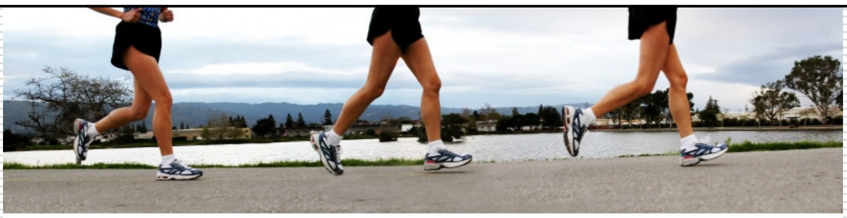
- Wicked problems characteristics
  - They are hard to state concisely
  - They can be symptoms of other problems
  - Solutions
    - have unforeseen consequences
    - are open to value judgments
    - can be hard to describe
    - may be hard to verify

Observations on solving wicked problems

- Time is required to let things “soak in”
- They either squarely demand your attention or lurk in your thoughts
- They are rarely solved by a committee
- Nearly impossible to predict when they will be solved

Agility and design problem types

- Sort work into “problem buckets” making sure each iteration gets enough core work accomplished
- Track how much time is spent on “the rest”
- Use post-iteration reflections to ask why things were harder than they first appeared
- Break out of planned iteration cycles to tackle revealing problems (they’ll need more than just a design spike)
- Make sure the team gets involved on core design issues



Sort Stories into Buckets

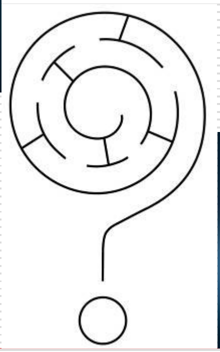
- We are planning for the first release
  - We must be able to **receive data from sensors**, **convert that data into normalized measurements** and **store them in the database**
  - We must **keep track of physical sensors, their location, operational status, and physical characteristics**
  - We must **predict fire danger rating**
- What seems core? What is less interesting?

Group Decision Making

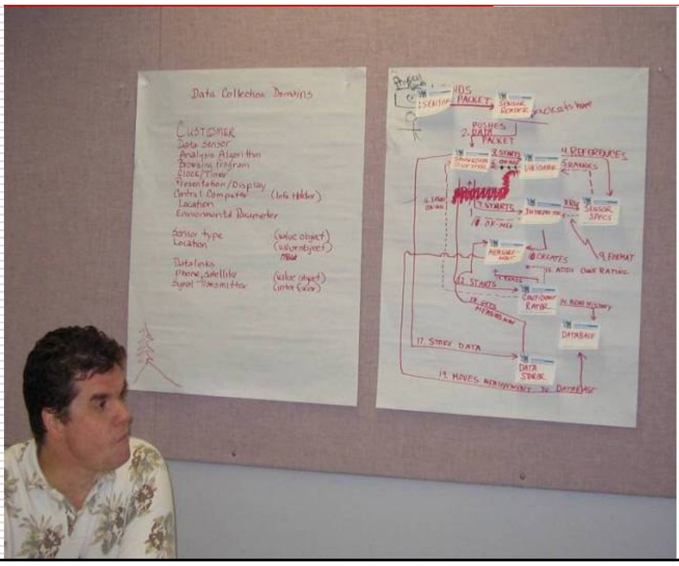
- Common ways to decide don’t always contribute to a design’s integrity
  - Voting
    - …Beware of the democratic fallacy
  - Dictatorship
  - Reaching consensus
  - Gathering
  - Sub-committee

A Two-Stage Decision Process

- When weighing options, a common approach goes something like this. After gathering your options:
  - Sort through them rapidly: No, no, no, maybe, no, no, no, no, maybe
  - Examine remaining options carefully



Handling Criticism



Valid

Not Valid

Aesthetics

Judgmental

Complexity

Personal

Great

Appropriate responses…

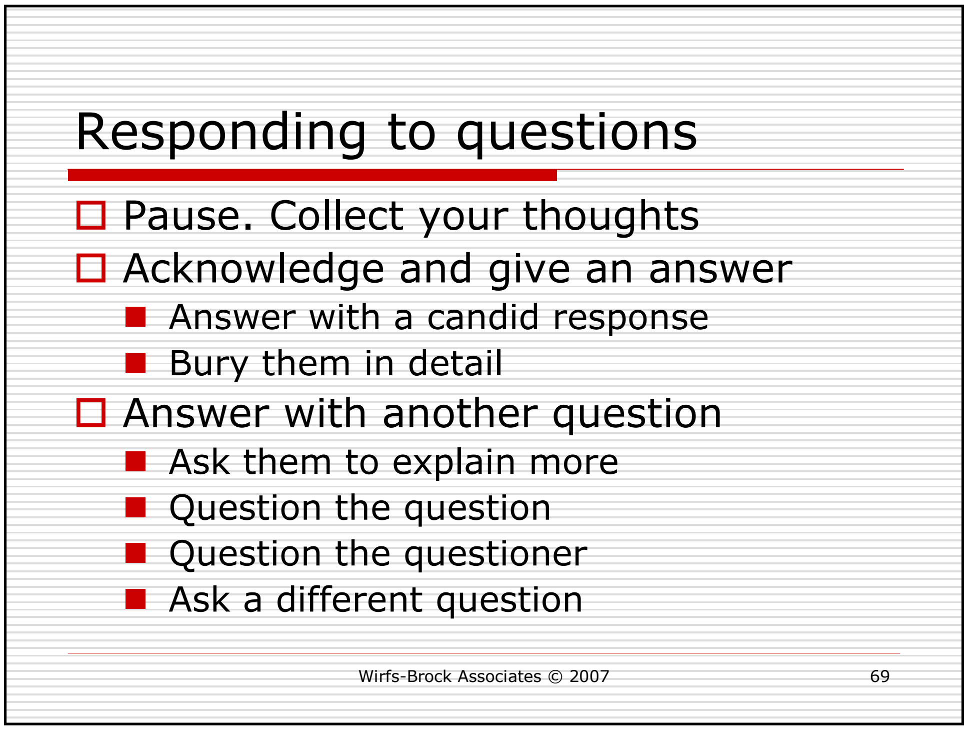
Type of criticism	Characteristics of criticism	Appropriate Tactic
Valid	Info indicates a flaw or weakness in idea	Refine your idea—but don't lose its advantages
Not valid	Clear misfit between your idea and criticism	Improve your ability to explain
Aesthetic	Negative reaction reflecting form vs. substance	Acknowledge, defuse by explaining your position
Judgmental	Negative reaction with/without enough info to indicate a problem	Ask critic for more specific info
Complexity	Value judgment with implicit assumption that a simpler solution exists	Explore. May need to educate about inherent complexity
Great!	May or may not be judgmental/specific	Optionally, probe behind the praise

Probing Questions

- Clarification…what did you mean by
- Purpose…why did you suggest that
- Relevance…does this apply here
- Completeness…is that all
- Accuracy…is that so
- Examples…can you give an example
- Extension…tell me more
- Evaluation…how good do you think it will be

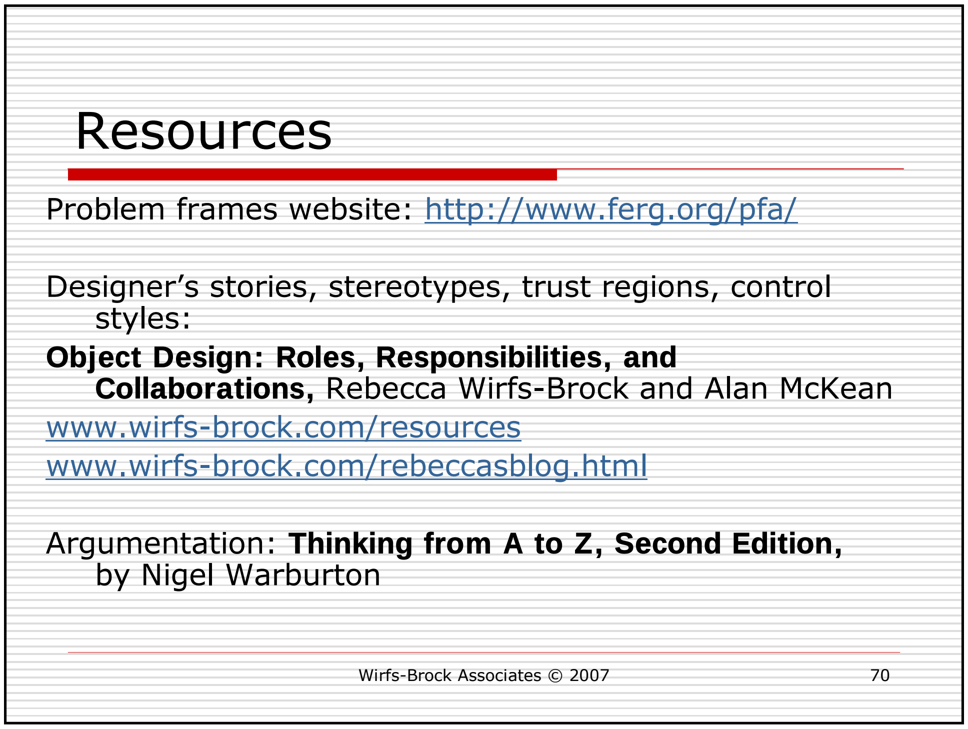
Clarifying Questions

- Get them to think:
  - Why do you say that?
  - What exactly do you mean?
  - How does this relate to what we discussed earlier?
  - Can you give me an example?
  - Are you saying ... or ... ?
  - Can you restate your concern?



Responding to questions

- Pause. Collect your thoughts
- Acknowledge and give an answer
  - Answer with a candid response
  - Bury them in detail
- Answer with another question
  - Ask them to explain more
  - Question the question
  - Question the questioner
  - Ask a different question



Resources

Problem frames website: http://www.ferg.org/pfa/

Designer’s stories, stereotypes, trust regions, control styles:

Object Design: Roles, Responsibilities, and Collaborations, Rebecca Wirfs-Brock and Alan McKean

www.wirfs-brock.com/resources

www.wirfs-brock.com/rebeccasblog.html

Argumentation: **Thinking from A to Z, Second Edition,** by Nigel Warburton

-Rebecca

rebecca@wirfs-brock.com

www.wirfs-brock.com

Skills for the Agile Designer
Supplementary information and notes

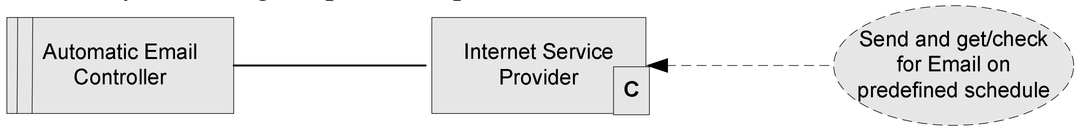
Rebecca Wirfs-Brock
rebecca@wirfs-brock.com

Tools for Seeing: Problem Frames

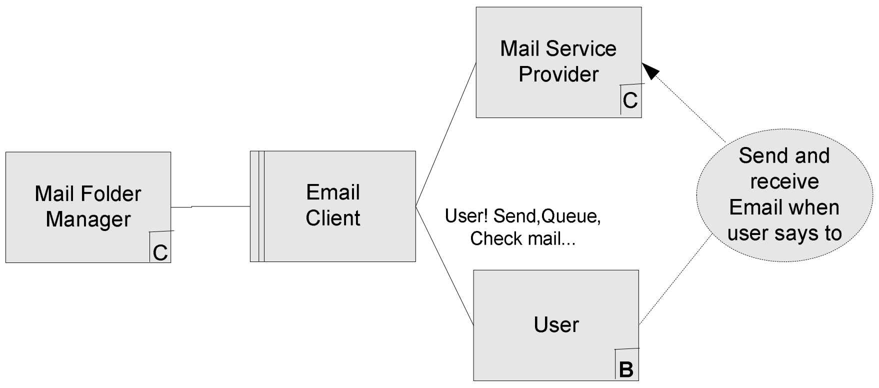
Problem frames are a way of mentally dividing your software's purpose into manageable chunks. Software systems can be thought of as a set of sub-problems or "problem frames". By breaking down a problem into its constituent problems, you can consider a large system one smaller piece at a time. Michael Jackson, who invented the notion of problem frames, writes about them in *Problem Frames: Analyzing and structuring software development problems.* Jackson suggests that because software serves so many purposes, developers should start by describing and structuring their problems in a way that, according to Jackson, is "rarely necessary in other engineering disciplines, where the diversity of problems to be solved is much smaller." Agile designers must become adept at asking: What kind of problem is this? What is our software all about? What purpose does it serve? What behavior and properties must our software have to achieve that purpose?

Each different class of problem frame has specific concerns and issues. When you think about a problem, if you can "fit your problem (or a piece of it) into a relevant frame" then it will lead you to ask appropriate questions and make appropriate tradeoffs. Here are definitions of Jackson's five problem types or frames and example frame diagrams:

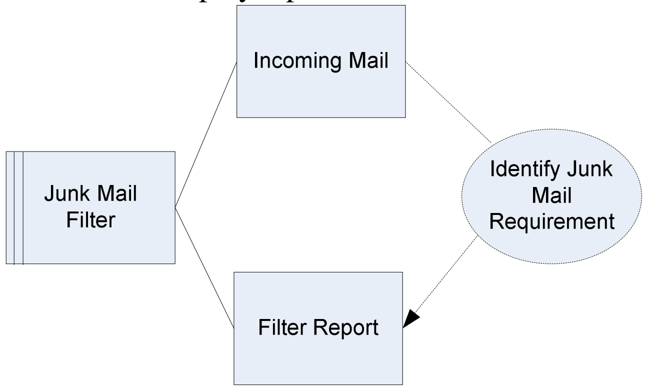
- Required behavior—controlling state changes of something outside your software machinery according to specific requirements.


Automatic Email Controller
Internet Service Provider
C
Send and get/check for Email on predefined schedule

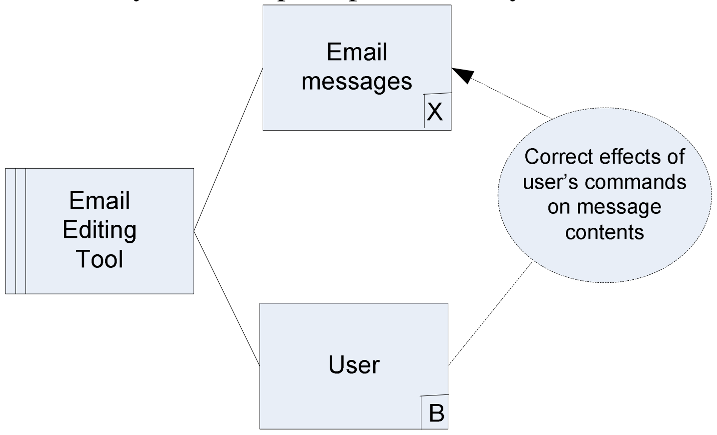
- Commanded behavior—controlling changes based on an operator or user's commands


Mail Service Provider
C
Mail Folder Manager
C
Email Client
User! Send,Queue, Check mail...
Send and receive Email when user says to
User
B

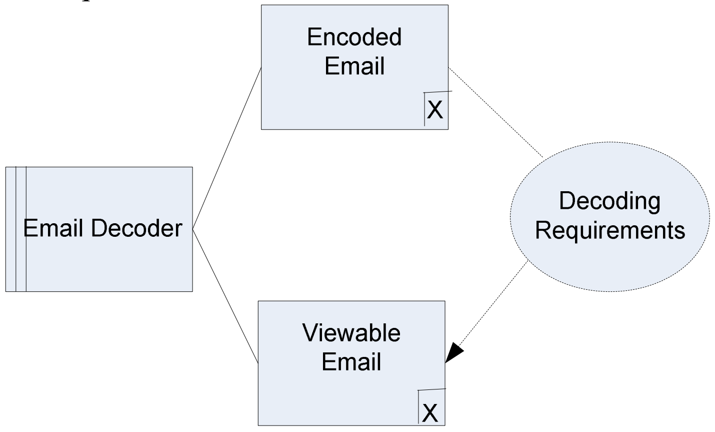
- Information display—produce information about some observable phenomena



- Simple workpieces—a tool that allows users to create and manipulate structures, so that they can be copied, printed, analyzed, or used



- Transformation—convert input to one or more outputs according to specific requirements



Frame diagrams are just a convenient iconic way to represent the structure of a problem, but don't worry they are secondary to the real value of framing—a thinking tool to help you gain understanding and focus your requirements and design. If you understand the nature of the problem your software needs to address, you can ask relevant questions that help shape and focus your work. You can use problem framing without adopting Jackson's formal approach.

In agile development there are a number of areas where framing is useful:

- To initially brainstorm what kinds of design challenges will predominate and what parts of your software they are likely to impact. In all but the simplest system there are usually multiple problems (and frames) that are evident. Problem framing is a good way to get teammates acquainted with upcoming design work and identify the potentially hard parts.
- As you discuss specific user stories with your customer. While I don't even mention problem frames to customers, I keep them in mind as we discussing any issue. I use them as a mental tool to sharpen my thinking. If you think about which problem frame is relevant (and what concerns there are) you will find yourself asking questions that buy you more information. And you can have more meaningful discussions with your customer about what your software should or shouldn't do.
- To assess additional work during a design spike. As you dig deeper into implementation you need to rethink and occasionally reframe the problems you are solving. A design spike happens whenever something is more complex than you had thought. It could be that reframing the problem might bring clarity.

When discussing what the system should do (and how things are) with you customer, it can be useful to distinguish truths or facts (indicative qualities) from desired (optative) behavior which is often imprecisely expressed as statements beginning with "shall" or "should".

Frame Concerns. Your goal is to design and build software that will behave appropriately and solve the customer's problem. Jackson advocates that you convince yourself and your customer that your proposed software will tackle the right problem by writing an appropriate set of descriptions about the problem domains. As a problem framer, your central task is to investigate and describe problem domain properties. Each class of frame has a different set of concerns that are typically addressed.

This is one area where Jackson and agile developers diverge on their approaches (and value equation). While I may advocate for formal descriptions *when they add value*, I find Jackson's insistence on writing descriptions of various domain properties to be a difficult task for most developers whether agile or not. I find these formalisms to be less valuable than knowing what questions to ask and what issues are commonly encountered in particular problem frames. So instead of going formal, I find myself asking probing questions about a particular frame. Once I've framed a problem, I can start asking questions. Or conversely, as I am asking questions I'm exploring what frames seem to fit and push harder to gather appropriate requirements.

Following are some stylized questions I've come up (consider them proto-questions) to ask about each type of frame as you are digging for understanding.

Questions to ask about a workpiece frame:

- What are the basic elements of the workpiece?
- Will it take different forms?
- Does it need to be shared? If so, how?
- Does it have an interesting lifecycle (or is it just something that is changed and then treated as "static" after each change?
- Is it passed around between various users? Is there a workflow associated with a workpiece?
- Should it persist? In what forms? Should it be published or printed?

Here are some questions to ask about required behavior problems:

- What external state must be controlled?
- How does your software find out whether its actions have had the intended effect? Does it need to know for certain, or can it just react later (when the state of some thing is not as expected)? What should happen when things get "out of synch" between your software and the thing it is supposedly controlling?
- How and when does your software decide what actions to initiate?
- Is there a sequence to these actions? Do they depend on each other?
- Are there complex interactions with your software and the thing under its control?
- Can you view the connection between your software and the thing under control as being direct (easier) or do you have to consider that it is connected to something that transmits requests to the thing being controlled (and that this connection can cause quirky, interesting behavior)? If so, then you may need to understand the properties of this "connection domain" that stands between your software and the thing being controlled?

Here are some questions to ask about transformation problems:

- What data do you start with?
- How will it be changed?
- Is the transformation complex?
- Will it always work? What should happen when you encounter errors in the input?
- Is the transformation "lossy" or reversible?
- What speed, space, or time tradeoffs are there for performing any transformation?

Here are some questions to ask about commanded behavior problems (in truth these are only the tip of the iceberg):

- What's a good model of user-system interaction?
- What does the user need to know in order to "command" the system to do things?
- Do certain commands need to be inhibited based on the current state of the system? Do they always make sense? Does a sequence of actions make sense?
- Is there a lag between issuing a command and the system performing the action? Is that a problem?
- What happens when a command fails? How should users be involved in "steering" the software when a command fails?

- Should certain commands be ignored (e.g. how many times do you need to press the elevator button to call the elevator to your floor)?
- Do commands need to be reversible? logged? monitored or otherwise tracked?

Here are some questions to ask about information problems:

- What is the form of "observation" that the software must make about some event or fact or thing? Is it difficult to ascertain when an event has occurred? (For example, if your software is trying to record how many "vehicles" passed over sensors place on the road it may be very difficult to characterize what constitutes a vehicle—is it two axles passing within a time period, but what about motorcycles, backed up slow traffic, etc., etc.)?
- How precise does the information need to be? Is the information "fuzzy"?
- How much computation does your software have to do to come to an observation? (For example, consider assigning a "junk mail rating" to an email, based on Bayesian analysis of the contents of the current message based on sample data currently loaded into the junk mail box?
- Is the user only interested in current information? Or is historical information important?
- Are there questions that the user may want to ask about the information? What are they? How easy are they to accurately answer?
- Does your software need to construct a "model" of the phenomena being observed in order to answer questions about it?

Tools for Seeing: A designer's story

A designer's story is a way for you to put your own spin on the system you are working on and a substitute for XP's elusive metaphor. Early on in any project I now write a designer's story. Originally, I used a design story as a private way to organize my thoughts. Lately, I've been encouraging teams to individually write designer's stories and then share them at the beginning of a project. This has been a good way to voice individual visions that can complement and be melded into a shared perspective. And it's a good ice-breaker for newly formed teams or in situations where some voices dominate and others' voices don't get heard.

Here are four reasons fro writing a designer's story:

- To restate any requirement from your design perspective
- To put your own spin on what's important or hard or easy or similar to what you've done before
- Boiling it down helps you grasp the problem
- To own the problem

Sharing your design stories with teammates allows you to:

- Have a voice
- Get others' perspectives
- Develop collective thoughts
- Build mutual understanding and trust

Technique: Write a designer's story.
The technique is very simple. Even those who only want to write code can bang out a story if it is short, sweet, to the point, and only take 15 minutes. I tend to pump out lots of words when I am put in front of a word processor. So I prefer to write my stories by hand, especially when I want to share them with others. This makes them more personal and shorter. They look rough and less polished which is a good thing.

A designer's story should be short—two paragraphs or less is ideal. Write about your application's essential characteristics: the themes. Take about important ideas such as:

- What is your application supposed to do? How will it support its users? Is it connection to a real world example that you can study or emulate? Is it similar to what you have done before?
- What will make you application a success? What are the most challenging things to design?
- What don't you know that you'd like to.

Tell what you know and what you need to discover.
Designer's stories are different than user stories or even descriptions of design objects. They are your impressions of your software.

Here is an example of an online banking application I worked on with 10 others. It was a system built for a consortium of South American banks. After reading the "spec" that the technical architect wrote after he came back from South America, I sat down and wrote a story to wrap my head around the system (after all I was the project leader and had to "own" the problem and the ensuing design). I never shared this story with my teammates and I was chatty as I wrote it in a word processor. I'll only show an excerpt:

> "This application provides internet access to banking services. It should be easily configured to work for different banks. A critical element in the design is the declaration of a common way to call in to different backend banking systems. We will define a common set of banking transactions and a framework that will call into banking-specific code that "plugs into" the standard layer implementing the details. The rest of our software will only interface with the bank-independent service layer…At the heart of our system is the ability to rapidly configure our application to work for different backends and to put a different pretty face on each. This includes customizing screen layouts, messages and banner text. The online banking functions are fairly simple: customers register to use the online banking services, then log in and access their accounts to make payments, view account balances and transaction histories, and transfer funds…"

Technique: Identifying application themes. Although you could stop after merely writing and sharing stories, I've found it useful to use them as a source of inspiration for identifying key aspects or important areas of design focus. I substitute "theme harvesting" when I cannot find an elusive metaphor to guide my design.

The themes I pulled from the online banking story were:
- Modeling online banking activities
- Representing common bank functions
- Configuring system behavior
- Access scarce resources (that was in the elided part of the story)

Themes can be broad or narrow depending on how you write your story. If you get into nitty gritty details, your story may be centered on one particular aspect. If they are too narrow, there may not be many objects to harvest, either. The broader a theme is, the more work it takes to drill down to an appropriate level to identify candidate objects. The idea is to hunt for an initial bunch of objects by mining a story's themes.

Technique: Leveraging themes to identify key areas of activity and initial candidates. Once you have identified major themes, you can use them (and your stories or story cards) as one source of inspiration. Make educated guesses about the kinds of inventions you'll need in your design based on the nature of your application and the things that are critical to it. I consider any candidates I harvest out of this brief dip into the system as "seed corn" that starts up my design thinking.

To hunt for candidate objects, consider each of these perspectives for any theme:
- The work your system performs
- Things affected by or connection to your application (other software or physical devices)
- Information that flows through your software

- Decision making, control and coordination activities
- Structures and groups of objects
- Representations of real-world things your application needs to know something about

If you find that a particular perspective doesn't yield any insights or ideas, move on.

For example, for the theme "online banking functions", considering the work our system performs led us to consider candidates that specifically supported performing financial transactions and querying accounts. Lots of information flowed through our system to accomplish banking functions—information about transactions, accounts, account balances, transaction amounts, account history, payments….many of these ended up as domain objects.

Identifying candidates that support each theme is a quick brainstorming activity. Sometimes candidates readily pop out when you look at a perspective. Often different themes and perspectives reiterate and reinforce certain candidates. This is good. It builds confidence in a candidate's relevance. At other times, ideas do not come so quickly and you must stretch your thinking to come up with potential candidates. You won't find all important candidates in this first pass look through your system and your ideas will certainly change—but it's a start.

In a brainstorming session a team can work up a candidate list in a couple of hours.

Tools for Seeing/Shaping: Concept Forming

According to Richard Fobes in *The Creative Problem Solver's Toolbox,* concept thinking is "yet another alternative to thinking in words." He suggests and I have found this true in my own design experience, that "it can be useful to become aware of a promising abstract concept before translating that abstract concept into something specific and concrete."

Concepts may be eventually become mechanisms, algorithms, ways of doing something, or eventually one or more "objects" in object design exist separately from words. In the slide for this tutorial I'm showing a diagram that graphically says, "Meat is to a vegetarian as what drugs are to an unnamed entity." That entity surely can be described in words "Someone who abstains from taking drugs for a variety of social, moral, or ethical reasons"….but you don't have to put a single word to that concept to understand it. (I know, those clever folks among you will be trying to come up with a single word—teetotaler, ex-junkie, …?, but that's not the point.

It still can be a good design concept or idea, even if you can put a neat, tidy single word name to it. Useful concepts you incorporate into design solutions can literally come from anywhere. I think this is in the spirit of what Kent Beck has talked about as "finding" the right metaphor. Point is, when you need to invent some new thing in your software, you can get inspiration by recognizing similarities between challenging problems that have been solved before and something that's seemingly unrelated.

It's interesting to note that some really creative ideas had their roots in making connections between unrelated things—for example the idea for typewriter keys came from watching a musician play an organ. If you can see similarities in what's different, then you can come up with a new concept.

To express unnamed concepts in words you can use examples or analogies.

For example, on one project, we envisioned a tool that "sliced" the right amount of interest based on the loan's amortization parameters. In fact, for want of a better word for a while we called this process a "slicer". And I've visualized account entries (and balancing of transactions) as analogous to balls (entries and counter entries) bouncing around in a pachinko machine of a network of accounts.

Tools for Seeing/Shaping: Object Role Stereotypes

Agile designers need to see and describe their ideas to others. If you all share the same way of talking about your design inventions and objects, then you'll improve how you communicate. **Role stereotypes,** from Responsibility-Driven Design are a fundamental way of seeing objects' behaviors. A stereotype is a "purposeful oversimplification" that you can use to identify the gist of an object's behavior. Later on you can stereotype objects or classes to characterize your implementation or somebody else's design:

Here is a synopsis of six stereotypes:

- Information holder—knows and provides information
- Structurer—maintains relationships between objects and information about those relationships
- Service provider—performs work and in general offers services
- Controller—makes decisions and closely directs others' actions
- Coordinator—still makes decisions, but primarily delegates tasks to others and keeps out of the way (there's a spectrum of behaviors from overly-dominating controller to laissez-fair coordinator)
- Interfacer—transforms information and requests between distinct parts of a system. There are user interfacer objects, for example, and external interfacers that may wrap other systems and objectify their services. But interfacers can be go-betweens from layers or subsystems, too.

Technique: Stereotyping a candidate

Can an object have more than one stereotype? Sure. Each candidate fits at least one. They often fit two. Especially if you are following the design principle of "making objects both know and do things". Common blends: service provider and information holder, interfacer and service provider, structurer and information holder. Identify the major stereotype you want to emphasize and check your initial ideas against your current implementation from time to time.

Technique: Identifying a candidate's purpose

I write a purpose statement on the unlined side of a CRC card. Not surprisingly, the candidate's purpose matches its stereotype. A candidate does and knows certain things. Briefly, say what those things are. A pattern to follow:

> An object is a type of thing that does or knows certain things. And then mention one or two interesting facts about the object, perhaps a detail about what it does or who it works with.

Here's a concrete example of a purpose statement:

> A FinancialTransaction represents a single accounting transaction performed by our online banking application. Successful transactions result in updates to a customer's accounts (that was to distinguish financial transactions from Queries that had no affect on account balances and didn't require the same audit trail)

What do you do with a purpose statement? It can be recycled into a class comment, once you believe a candidate will stick around.

Technique: Identifying responsibilities
Whether an object primarily “knows things”, “does things”, or “controls and decides” is based on its role stereotype. Exploring an object’s character will lead to an initial set of responsibilities. For example, information holders answer questions. They are responsible for maintaining certain facts that support these questions. Rather than listing out all the “attributes” of an object, or going into details about its variables, responsibilities are a higher level view of an object. Instead of talking about a customer’s first name, last name, surname, nickname, etc…. you can state this general responsibility as “knows name and preferred ways of being addressed.

When designing a service provider as “what requests should it handle”? Then, turn around and state these as general statements for “doing” or “performing” specific services. Again, responsibilities can be written at a higher-level than a single method or operation. For example, you can talk about “compares to other dates” instead of listing out “>”, “<”, “<=”, etc.

Shaping Tool: Control Center Design

Deciding on and developing a consistent control style is one of the most important decisions designers make. Agile designers can benefit from a vocabularly to describe their design choices. A **control center** is a place where objects charged with controlling and coordinating reside.

Developing a control style involves decisions about:

- How to control and coordinate application tasks
- Where to place responsibilities for making domain-specific decisions, and
- How to manage unusual expected conditions (the design of exception detection and recovery)

While it is true that many frameworks make some of these decisions for you, there is much room for judgment (and lots of options to explore). It isn't just a matter of style. Control design affects complexity and ease or difficulty of your design to change. Your goal should be to develop a dominant, simple enough pattern for distributing the flow of control and sequencing of actions among collaborating objects.

A control style can be **centralized, delegated,** or **dispersed.** But there is a continuum of solutions. One design can be said to be more centralized or delegated than another.

If you adopt a centralized control style you place major decision-making responsibilities in only a few objects—those stereotyped as controllers. The decisions these controllers make can be simple or complex, but with more centralized control schemes, most objects that are used by controllers tended to be devoid of any significant decision-making capabilities. They do their job (or hold onto their information), but generally they are told by the controller how to do so.

If you choose a delegated control style, you make a concerted effort to delegate decisions to other objects. Decisions made by controlling objects will be limited to deciding what should be done (and handling exceptions). Following this style, objects with control responsibilities tend to be coordinators rather than control every action.

Choosing a dispersed control style means distributing decision-making across many objects involved in a task. I haven't worked on systems where I've consciously used this style, although you could consider a pipes-and-filters architecture or chain-of-responsibilities patterns to be a dispersed control style.

Nothing is inherently good about any particular style. They all have plusses, minuses, and things to watch out for. But generally, I prefer a delegated control style as it seems to give more life (and responsibilities) to objects outside a control center and avoids what Martin Fowler calls "anemic domain models". In a nutshell, here are characteristics of each style.

Centralized control: Generally, one object (the controller) makes most of the important decisions. Decisions may be delegated, but most often the controller figures out what to do next. Tendencies to watch for with this strategy:

- Control logic getting overly complex
- Controllers becoming dependent upon information holders' contents
- Objects becoming indirectly coupled as a result of the controller getting information out of one object and stuffing it into another
- Changes rippling among controller and controlled objects
- The only interesting work (and programming effort) being done in the controller

Delegated control: A delegated style passes some of the decision making and much of the action off to objects surrounding a control center. Each neighboring object has a significant role to play:

- Coordinators tend to know about fewer objects than dominating controllers. They are easier to test.
- Message between coordinators and the objects they collaborate tend to be higher level requests (e.g. instead of setters and getters and minute calls, there are more "Nike" requests—justDoIt()).
- Changes are typically more localized and simpler
- Easier to divide interesting work among a team

Dispersed control: A dispersed control style spreads decision making and action among objects that individually do little, but collectively, their work adds up. This isn't an inherently bad strategy; but avoid these tendencies:

- Little or no value added by those receiving a message and merely delegating to the next object in the chain
- Hardwiring dependencies between objects in long collaboration chains

Technique: Control Center Design
Don't adopt the same control style everywhere. Develop a control style suited to each situation:

- Adopt a centralized style when you want to localize decisions in one place
- Develop a delegated style when work can be assigned to specialized objects
- Several styles can and should co-exist in a complex application
- Look at how a particular framework (or accepted style of programming, say, how a J2EE application "typically does things") impacts the control styles you adopt and whether it injects undue complexity into your design. For example, a style that separates business rules from information holder objects results in 2x the number of classes, but arguably makes it easier to unit test information holders.
- Assess whether your ideas about control style line up with other experts or pattern authors

Control styles within subsystems vary widely. But as a general design rule, make analogous parts of your design be predictable and understandable by making them work in similar ways.

Tools for Shaping: Trust Regions

One way to get a handle on where collaborations might be streamlined and simplified is to carve your software into regions where trusted communications occur. Generally, objects located with the same **trust region** communicate collegially, although they still encounter exceptions and errors as they do their work. Within a system there are several cases to consider:

- Collaborations among objects that interface to the user and the rest of the system (unless information it is verified before it is sent to the rest of the system, it shouldn't be trusted to be valid)
- Collaborations among objects within the system and objects that interface with external systems
- Collaborations among objects outside a neighborhood or subsystem and objects inside
- Collaborations among objects in different layers
- Collaboration among objects you design and objects designed by someone else
- Collaborations with library objects

When objects are in the same layer or neighborhood, they can be more trusting of their collaborators. And they can assume that objects that use their services call on them appropriately.

If a request is from untrusted or unknown sources, extra checks are typical before a request is honored.

When an object uses a collaborator outside of its trust region, it may take extra precautions, especially if it has responsibilities for making the system more reliable. It may need to:

- Pass along a copy instead of sharing data
- Check on conditions after a request completes
- Employ alternate strategies when an exception is raised

Objects at the "edges" of a trust region typically take on more responsibility. For example, an object receiving a request from an "outsider" may make initial checks, only passing along know good requests or data to others.

Design Collaboration: Handling Criticism

OK, you have ideas and you want to get them out there. Criticism of any new or novel idea is inevitable (whether justified or not). While it may be desirable to handle criticism in a dispassionate way, I used to find myself getting caught up in defending my ideas instead of learning from what others were saying. Sure, you've got to get buy in from your team, and because you value the wisdom of your team you want to listen to and respond appropriately to what they say. But not all comments should carry the same weight nor should they be treated the same way.

Knowing what tactic to take when someone levels a question or comment at an idea is invaluable to keeping your creative juices flowing, improving on your design ideas (and knowing when to let a comment roll off your back).

Here's a summary of the types of criticism you may receive and how you might react:

Valid Criticism- The person has pointed out a flaw or weakness in your idea. You can quickly see that they're right because they've included enough information in their comments. In this case, the best action isn't always to fold up your tent and give up your idea. Before doing that, you might want to see how you could refine your idea to handle the valid criticism. You also might want to ask them to give more details. Depending on the complexity of the problem or the nature of the criticism you may want to take some time to think about it before thinking or reacting with a counter solution. If the person is willing to brainstorm with you about that, great. But not all valid criticisms have to be immediately handled in real time. The most useful criticism to receive is specific valid criticism.

Judgmental Criticism- If someone responds to your idea with "that won't work" or "I don't like that solution" they're being judgmental. They have some valid criticism (or not), but until you get more information that allows you to determine whether they revealed some flaw in your idea…you can't do anything with a judgment. To counter their judgment you need to ask, "What part won't work?" or "Why don't you like that?" and then listen. You may find out that they have a valid criticism (or partially valid criticism) or not. At the point you understand the specifics of their criticism, then you can take action. If they can't articulate why, well, there's not much you can do except keep asking them clarifying questions.

Invalid Criticism- If someone is giving you a clearly invalid criticism it may be because 1) they have a different perspective than you do, or 2) they don't fully understand your idea. Your job, should you choose to take it on (and this is an option), is to figure out how to better communicate your idea. Maybe the person doesn't "get it" because he needs to see concrete examples, or maybe he needs to see a picture or rough sketch (instead of some quickly hacked code), or maybe he needs to see some code (instead of a quickly hacked picture), or maybe you need to write some tests to demonstrate…if you want to communicate, you need to learn how to explain it in terms that the person can understand. Not everyone has the same frame of reference as you do.

Personal Criticism- Instead of making a judgment about your idea, they're stating a judgment about you: "You're stupid." If this shakes your confidence, ouch! Personal integrity attacks shouldn't be tolerated and can really drag a team down. (It is outside the scope of this tutorial on how to handle those cases. The best immediate reaction is to not rise to the bait).

Aesthetic Criticism- Someone has just leveled a comment that indicates he's coming from a different perspective than you and finds your solution to "not fit with his idea of good". There are many acceptable different ways to solve a design problem, and which one among several solutions may arguably be a toss-up. If you want to keep your idea alive, there's no need to cave in to aesthetic arguments. Just let them go. If you are responsible for solving that part, then your aesthetics should be permitted and encouraged. If aesthetic arguments persist, this usually it is a sign of egos clashing that needs to be addressed before the team can really establish trust.

Compliments- Someone just said your idea was good. Or great. There's not much you can learn from those kind of compliments. In fact, I find if people say, "great" too often, it may mean they are disinterested, too busy, haven't thought things through, or just don't feel comfortable enough to level a constructive useful criticism because of how you react (you have to be in tune with both them and the situation to make a call on why).

For another look at the types of criticism you may receive, read a recent IEEE Software Design Column, *Handling Design Criticism,* I wrote. I added another type of criticism that is common on agile teams (your design is too complex) and talk about how to address that kind of criticism.You can find a copy on my website: http://www.wirfs-brock.com/PDFs/handlingcriticism.pdf

Design Collaboration Tool: Argument Moves

A designer, especially one in an agile team, has to be a good communicator. Part of being a good communicator means knowing how to tell a sound from an unsound argument, and then knowing techniques for countering certain arguments. By the way, argumentation isn't the same as "shouting" or "having a fight". When I'm talking about argumentation, I'm talking about having a discussion on some topic. What's important to spot is when someone introduces faulty reasoning into an argument that momentarily throws you off track. The following are definitions of moves in an argument drawn from *Thinking from A to Z* by Nigel Warburton.

1. Red herring: Deliberate introduction of irrelevant topics into discussion. A red herring is literally a dried fish that when dragged across a fox's trail leads the hounds to chase the wrong scent. Introducing a red herring particularly effective because it may not be obvious at first that the trail is a false one, and red herrings are intrinsically interesting and may appear to be relevant.

2. Correlation = cause confusion: Two events may be correlated (that is, when one is found, the other is usually found) without there being a direct causal connection between them. Nevertheless many people act as if any correlation provides evidence of a direct cause and effect. Correlations may stem from coincidence rather than causal links.

3. Contraries: Two statements which cannot both be true, though they can both be false. This is especially relevant to consider when examining claims about goodness. For example, two vendors make claims to be "the best" at something, when in fact, neither may be the best.

4. Proof by ignorance: Where lack of known evidence against a belief is taken as an indication of it being true. For example, just because no one has provided conclusive evidence that ghosts exist (or gremlins that cause bugs in our software) it would be extremely rash to treat this as proof that they exist. Part of the temptation to believe that proof by ignorance is real proof may stem from our legal system, where a defendant is presumed innocent unless proven guilty.

5. Slippery slope argument: A type of argument which relies on the premise that if you make a small move in a particular direction it will then be extremely difficult to prevent a much more substantial move in the same direction. For example, the belief that if Oregon permits doctors to issue life-ending prescriptions to the terminally ill, then euthanasia of the aged or disabled without their consent will be even closer to happening. This form of argument can have some force, but in order to judge it we need information about the alleged inevitability of the descent; it is not enough to claim there is a slippery slope. Typically, slippery slope arguments obscure the fact that in most cases we can decide how far down a slope we want to go.

6. Pseudo-profundity: Uttering statements which appear deep, but are not. One of the easiest ways to generate pseudo-profound statements is to speak in paradoxes. For example: "Knowledge is just another form of ignorance. Shallowness is an important kind of depth. Effective resource allocation is a matter of freeing up the computer." Another way to achieve pseudo-profundity is to repeat banal statements: "Computers help people compute." A third way of generating pseudo-profundity is to ask strings of questions and leave them hanging in the air: "Will we ever have a source code repository? Can we ever keep source code up to date?"

7. False dichotomy: A false dichotomy occurs when someone sets up choices so that it appears there are only two possible conclusions when in fact there are further alternatives not mentioned. Most of the time the phrase "if you're not for us you must be against us" is a false dichotomy. There is another possibility, that of being totally indifferent to the idea, and yet another, that of being undecided.

8. Shifting the goal posts: Changing what is being argued for in mid-debate. This is a common move to avoid criticism: as soon as an arguer sees a position becoming untenable, he or she shifts the point of discussion to a related, but more easily defended one.

:

9. Companion in guilt move: Demonstrating that the case in question is not unique. This is usually intended to dilute the force of your argument by showing that demands of consistency should lead you to apply the same principles to further cases, something that you may not want to do. When encountering a the companion in guilt move, you may be forced to be explicit about what you take to be unique to the topic in question.

10. Democratic fallacy: The unreliable method of reasoning which treats majority opinion as revealed by voting as a source of truth and a reliable guide for action on every question. There are many areas where taking a vote would be an extremely unreliable way of discovering the most appropriate course of action, especially if the majority of voters are largely ignorant of the matter.

What are some ideas you have on ways to effectively head off or counteract one of these moves when appropriate?

Design Collaboration Technique: Sorting your design work

As a designer, you are expected to be a good problem solver. You can be prepared with a toolkit full of design techniques and practices, but design is never predictable. There are always surprises, additional complexity, and new twists. To keep on track, fit your development tasks into these categories:

- Core design problems—the core is the core because without it there is no reason to build the rest. You application won't meet its users' needs without a well-designed core. Core design problems absolutely, positively must be dealt with.
- Revealing design problems—when pursued, these problems lead to a deep understanding about your software. Just because some part of a design is tricky or difficult, however, doesn't make it revealing.
- The rest. Although not trivial (well, not all the time), the rest requires far less creativity or inspiration.

Each task warrants a different approach and has a different rhythm to its solution. Core problems must be solved. You've got to give them proper attention. Revealing problems are squishy and hard to characterize or even know when they are solved. Each time you look into a revealing problem it teaches you something. Revealing problems can't be solved in tidy ways—they must be tamed. But the rest can't be ignored either. It is always present and pressing. If you don't budget your time, it can soak up all spare cycles.

Technique: Sort design tasks into buckets
At the start of each iteration or sprint, some teams sort their tasks into "core" and "the rest". This can help you pick tasks to work on from a backlog.

Depending on your design, you might nominate as core:

- Key domain objects
- Design of important control centers
- Key algorithms
- Mechanisms that increase reliability such as exception handling and recovery, synchronization and connection with other systems, performance tuning, caching,…

To decide whether something is core ask what are the consequences of "fudging" on that part? Would the project fail or other parts of your design be severely compromised? Then it's core. When a team disagrees about whether certain tasks are core or not, dig deeper. It may be that someone isn't getting listened to (so in order to be heard they want to elevate the importance of particular tasks that they find comfortable or familiar). Or, it may be that you don't listen to them (and they may have something to teach you). Whether you classify something as "core" or "the rest", you'll still have to deal with it—it's just a matter of emphasis. In any iteration give design tasks the attention they deserve and be clear on the team's priorities.

At the end of a sprint you may want to sort through your work in a reflection and judge, well, which tasks are soaking up time (but they aren't core), which things are not getting the attention they deserve, and what should be done about them.

You also may want to assign core tasks as "paired tasks" (e.g. requiring that two heads look at core problems), but any of the "rest" may be done solo.

Resources

Problem Frames

Last year I wrote a paper with Paul Taylor and James Noble that was accepted at the patterns conference (PLoP) that described Problem Frames in pattern form. You can download a copy of this from my website.

> *Problem Frame Patterns*, Rebecca Wirfs-Brock, Paul Taylor, and James Noble, PLoP 2006
> available at http://www.wirfs-brock.com/PDFs/ProblemFramePatterns.pdf

This is the definitive book by Jackson on Problem Frames. Be aware that it contains formal descriptions of events and shared phenomena between domains, state diagrams, as well as a definite slant towards software machinery interacting with physical domains in the real world. I had to get over this bias before I could start framing software intensive systems' frames.

> *Problem Frames: Analyzing and structuring software development problems,* Michael Jackson, Addison-Wesley, 2001

A website devoted to Problem Frames and the Problem Analysis approach. You can find links there to articles and papers that have demonstrated the use of problem framing in requirements analysis. But be aware. The general belief held by this site (and Jackson) is that Problem Frames and XP practices don't mix very well. I agree that if you take problem frames and equate them with formal descriptions, they don't mix well. But if you use framing as a questioning technique, they do.

> http://www.ferg.org/pfa/

Designer's Stories, Role Stereotypes, CRC Cards, Trust Regions, Control Styles, Design Problem Types

This book, in addition to my website resources page contains the latest on various techniques and ways of seeing that are part of Responsibility-Driven design practices. In the book, we call them "design stories" but I like "designer's stories" name better. This is to avoid confusing designer's stories with user stories and story cards in XP development:

> *Object Design: Roles, Responsibilities, and Collaborations,* Rebecca Wirfs-Brock and Alan McKean, Addison-Wesley, 2003
> **http://www.wirfs-brock.com/Resources.html**

Argument Moves

This handy little book is a lexicon of argument moves. It is easy to read, well written, and only costs $12.21 on amazon.com.

> *Thinking from A to Z, Second Edition,* Nigel Warburton, Routledge Taylor & Francis Group, 2000

Creative Problem Solving

The Creative Problem Solver's Toolbox, by Richard Fobes is an interesting read. I gleaned two ideas from that book for this tutorial: that of concept formation and the other about how to creatively handle criticism. If you are looking for other ideas on how to foster your own creativity, build new ideas or ways of looking at problems, this book might provide you some inspiration. To order the book, see www.galenpress.com

Data Collection Problem OOPSLA DesignFest™ Problem

Note: The following design problem description will be used in several exercises we'll do (the exact number depending on time permitted). This example is used with permission by OOPSLA DesignFest folks who, by the way, publish past design fest descriptions on their website (so that anyone can use them in classroom or training settings)…I've made a few technology modifications to reflect a more modern world.

Background

A local forest technology company, Forests 'R' Us, wants to build and sell a system for gathering and analyzing weather information to predict forest fires and help with water table management. The Arbor2100 will be sold to National Forests, Environment Canada, the U.S. Forest Service, and large private landowners. It will consist of hardware and software both locally in the owner's office building and remotely in the forests.

The data sensors in the forest report at various intervals to our system computer via satellite, packet radio, cell phone, or dedicated line. The system stores and analyzes the information. The users run a wide variety of reports, browsers, historical trend analysis, and future prediction algorithms over the data. Furthermore, given the inherently geographic nature of the data, many of the reports incorporate maps.

The sensors, such as temperature, sunlight intensity, wind speed and direction, rainfall, and so on, com in three basic types:

1. those that report on a regular basis (every minute, hour, day, month),
2. those that only report when a significant event occurs (a certain amount of rain has fallen, the temperature rises above a threshold), and
3. those that must be queried.

Some sensors fall in multiple groups; for example, they report events but can also be queried.

The sensors are produced by different manufacturers and return numeric values in a wide variety of units (miles/hour, km/hour, lumens, watts, calories/day, etc.) and at widely varying intervals and tolerances.

Additionally, not all data links are necessarily reliable, and yet the system must deal with all these issues while presenting both a uniform and a detailed view of the data to the user and his or her agent/analysis programs.

Desired Programs

Forests 'R' Us needs three categories of programs:

1. one to gather the sensor data as it arrives and store it in a database,
2. one to configure the field sensors, and

3. one to provide the user interface for browsing and analyzing the data.

Gathering the sensor data is relatively simple: the field sensors send information packets to the central computer, and the central computer stores them. Each packet contains a sensor ID, the time stamp, and the numeric sensor measurement. Some sensors may also report GPS coordinates (more modern ones) while older equipment may not. For cost reasons, many sensors are grouped into sensing units which send their data together (e.g., wind speed, direction, humidity, and temperature).

Configuring the field sensors consists of telling the software where each sensor is physically located and what type of sensor it is. Additionally, many sensors have different settings for measurement units and errors, reporting intervals, etc., so these too are configured. Because this is a 7 x 24 system, sensors can be replaced at any time, usually with an upgraded model and thus with different measurement units, error tolerances, etc.

The browsing and analyzing programs are the heart of the system. The analysis algorithms provide fire danger ratings, water table estimates, flash flood warnings, and so on. The browsing interfaces provide detailed information, both tabular and geographic, from the database. For example, the temperature maps similar to those seen on the evening news are one of the possible graphical outputs. The user should be able to navigate through the information in many ways including:

1. Map browsing multiple sensor types (temperature and rainfall) or multiple time periods (temperature over the previous month).
2. Browsing the type and status of the sensors at any location or locations.
3. Browsing the reliability and age of the information for any sensor and/or location.

To provide for future expansion, each of the predicted values available for display (e.g. temperature, rainfall, fire danger, flash flood risk, etc.) should be computed via a plug-in module. (Forests 'R' Us intends to sell additional modules for other risk factors, such as earthquake prediction, in the future.)

Common Situations
The following are typical scenarios and conditions that the Arbor2000 software is expected to handle.

Situation #1
There are sixteen sensor groups, each with three or four sensors, placed in the Rumbling Range National Forest. The sensors are randomly chosen from rainfall, temperature, sunlight, wind speed, wind direction, and snowpack sensors. The sensors report from once a minute to once a day and in a variety of units.

Jane Arden, a National Park Service Ranger, wants to post the fire danger results outside the Visitor Information Center, so she uses the Arbor2100 to examine the graphical view of fire danger in the forest. Overall, the fire danger is “moderate” with one area of “low danger + high uncertainty”. Looking into the uncertain area, she finds that a number of the sensors have not reported for quite a while, leading to the uncertainty. Further investigation reveals that none of the sensors in group 2 and 4 have reported, and further checking shows that groups 2 and 4 are the only two which use the 555-3473 phone modem. She dispatches a repair crew to figure out the problem with the phone line while she posts the “moderate” fire danger sign in front of the visitor’s center. She also checks the fire danger last year, and finds out that it was “low” over the entire forest, so she calls the Rumbling Range Spokesman-Review and asks them to print a story about how the fire danger is higher this year due to lower than expected rainfall.

Situation #2
The Rumbling Range National Forest buys two additional sensor arrays and hires a helicopter crew to plant them in the forest. After they return with Global Positioning System confirmation of the latitude and longitude of the sensors, Jane configures the system to receive the new data. Fortunately, the Arbor2100 is clever enough to store the unidentified incoming data until Jane had time to indicate where the arrays were located and what sensor types they were.

Situation #3
Forests ‘R’ Us comes out with a new plug-in module that it generously gives away free over the Internet. This new module computes trend analysis of the sunlight sensors to detect premature failure. Ms. Arden downloads and runs the module against the Rumbling Range database, only to discover that sensor #372 on Bald Mountain shows signs of age—its measured output has slowly declined over the past four years. Jane decides to hike to the top of the mountain and replace the sensor.

When she reaches the top, she discovers that the problem is not the sensor, but rather a small pine tree shielding the sensor from the sun. Unwilling to cut down the only tree on Bald Mountain, she relocates the sunlight sensor 100 meters to the south. When she returns to base, she updates the database with the sensor’s new location.