CM40212: Internet Technology

Julian Padget
Office: 1W2.15
E-mail: jap@cs.bath.ac.uk

Software Engineering
and
Web Architecture

Acknowledgements: Deitel & Deitel, Wikipedia

November 15, 2010
Outline

1. Agile Development
2. Design Patterns
3. Web Architectures
4. RIAs
5. Web Services
   - WSDL
   - SOAP
   - UDDI
   - REST
   - Workflows
6. Summary
Objectives

- Review principles of software engineering and link them to web development.
- To observe the cycle of history (mainframe, desktop, thin-client, rich internet application...)
- To note the steady increase in distributed and component-oriented application development
- And the re-application of old ideas (RMI in particular)
- But also the transience of all the software frameworks...
Objectives

- Review principles of software engineering and link them to web development.
- To observe the cycle of history (mainframe, desktop, thin-client, rich internet application...)
- To note the steady increase in distributed and component-oriented application development
- And the re-application of old ideas (RMI in particular)
- But also the transience of all the software frameworks...
Objectives

- Review principles of software engineering and link them to web development.
- To observe the cycle of history (mainframe, desktop, thin-client, rich internet application...)
- To note the steady increase in distributed and component-oriented application development
  - And the re-application of old ideas (RMI in particular)
  - But also the transience of all the software frameworks...
Objectives

- Review principles of software engineering and link them to web development.
- To observe the cycle of history (mainframe, desktop, thin-client, rich internet application...)
- To note the steady increase in distributed and component-oriented application development
- And the re-application of old ideas (RMI in particular)
- But also the transience of all the software frameworks...
Objectives

- Review principles of software engineering and link them to web development.
- To observe the cycle of history (mainframe, desktop, thin-client, rich internet application...)
- To note the steady increase in distributed and component-oriented application development
- And the re-application of old ideas (RMI in particular)
- But also the transience of all the software frameworks...
Content

1. Agile Development
2. Design Patterns
3. Web Architectures
4. RIAs
5. Web Services
6. Summary
Software Engineering

“In preparing for battle I have always found that plans are useless, but planning is indispensible” attributed to Beck and Fowler in Planning Extreme Programming

- What are the options?
  - waterfall – largely discredited
  - evolutionary – cost of throwing away
  - re-use – re-use what? re-factor!
  - formal – technically demanding
  - extreme (XP) – full-time activity ... adapt!
Software Engineering

“In preparing for battle I have always found that plans are useless, but planning is indispensible” attributed to Beck and Fowler in Planning Extreme Programming

- What are the options?
  - waterfall – largely discredited
  - evolutionary – cost of throwing away
  - re-use – re-use what? re-factor!
  - formal – technically demanding
  - extreme (XP) – full-time activity ... adapt!
Development patterns

- Origins? Goal of SE: engineering processes for s/w...BUT
  - specification often incomplete
  - phases (spec/design/manufacture) often overlap
  - no physical product to test
  - s/w does not wear out: maintenance $\Rightarrow$ modification

- Activities in the basic development pattern:
  - specification: definition of functionality and constraints
  - development: preparing design/writing program(s)
  - verification & validation: spec,design $\vdash$ program
  - evolution & maintenance: need + context change

- Agile integrates these processes using (essential) tools: e.g JUnit, PHPUnit, refactoring IDEs, FIT (Framework for Integrated Tests).
Development patterns

- Origins? Goal of SE: engineering processes for s/w...BUT
  - specification often incomplete
  - phases (spec/design/manufacture) often overlap
  - no physical product to test
  - s/w does not wear out: maintenance ⇒ modification

- Activities in the basic development pattern:
  - *specification*: definition of functionality and constraints
  - *development*: preparing design/writing program(s)
  - *verification & validation*: spec, design ⊢ program
  - *evolution & maintenance*: need + context change

- Agile integrates these processes using (essential) tools:
  - e.g JUnit, PHPUnit, refactoring IDEs, FIT (Framework for Integrated Tests).
Development patterns

- Origins? Goal of SE: engineering processes for s/w...BUT
  - specification often incomplete
  - phases (spec/design/manufacture) often overlap
  - no physical product to test
  - s/w does not wear out: maintenance ⇒ modification

- Activities in the basic development pattern:
  - specification: definition of functionality and constraints
  - development: preparing design/writing program(s)
  - verification & validation: spec,design ⊢ program
  - evolution & maintenance: need + context change

- Agile integrates these processes using (essential) tools: e.g JUnit, PHPUnit, refactoring IDEs, FIT (Framework for Integrated Tests).
Extreme Programming

- users write brief scenarios + estimate length of tasks
- schedule and negotiate/select most valuable
- team writes tests – clarify requirements/validate product
- do “precisely enough” design
- implement, test, refactor, repeat
- robust to requirements, design, personnel, etc. changes
- ...Scores well on process characteristics
Extreme Programming

- users write brief scenarios + estimate length of tasks
- schedule and negotiate/select most valuable
- team writes tests – clarify requirements/validate product
- do “precisely enough” design
- implement, test, refactor, repeat
- robust to requirements, design, personnel, etc. changes
- ...Scores well on process characteristics
Extreme Programming

- users write brief scenarios + estimate length of tasks
- schedule and negotiate/select most valuable
- team writes tests – clarify requirements/validate product
- do “precisely enough” design
- implement, test, refactor, repeat
- robust to requirements, design, personnel, etc. changes
- ...Scores well on process characteristics
Extreme Programming

- users write brief scenarios + estimate length of tasks
- schedule and negotiate/select most valuable
- team writes tests – clarify requirements/validate product
- do “precisely enough” design
  - implement, test, refactor, repeat
- robust to requirements, design, personnel, etc. changes
- ...Scores well on process characteristics
Extreme Programming

- users write brief scenarios + estimate length of tasks
- schedule and negotiate/select most valuable
- team writes tests – clarify requirements/validate product
- do “precisely enough” design
- implement, test, refactor, repeat
- robust to requirements, design, personnel, etc. changes
- ...Scores well on process characteristics
Extreme Programming

- users write brief scenarios + estimate length of tasks
- schedule and negotiate/select most valuable
- team writes tests – clarify requirements/validate product
- do “precisely enough” design
- implement, test, refactor, repeat
- robust to requirements, design, personnel, etc. changes
- ...Scores well on process characteristics
Extreme Programming

- users write brief scenarios + estimate length of tasks
- schedule and negotiate/select most valuable
- team writes tests – clarify requirements/validate product
- do “precisely enough” design
- implement, test, refactor, repeat
- robust to requirements, design, personnel, etc. changes
- ...Scores well on process characteristics
Time Boxing I

- Agile development uses **time boxing** rather than **feature boxing**.

- Given a fixed set of resources you can deliver either:
  - A fixed set of features in the time needed to realise these features—**feature boxing**
  - The number of features that can be realised in a fixed amount of time—**time boxing**

- To realise a fixed set of features in a fixed amount of time with a given set of resources is only possible if the time is sufficient to realise all these features.

- In practice, usually the time allowed is insufficient to realise all the features asked: *What the customer wants, he cannot afford.*

- Trying to do the impossible wastes project energy.
Time Boxing II

- In practice, time boxing means:
  - A set period of time is allowed for the task
  - At the end, the task should be 100% done.
  - Time slip is not allowed, otherwise the other tasks committed to in this cycle may not all be done.
  - Before the end, check how much will be achieved.
    - If task cannot be finished, define what is known, what needs investigation, estimate time for investigation.
    - Check if task can be completed in less detail.
    - Add new tasks to work list.
Content

1. Agile Development

2. Design Patterns

3. Web Architectures

4. RIAs

5. Web Services

6. Summary
Design patterns

- Object-oriented software engineering has developed a rich language for control models

- **Design patterns** = abstract form of re-use:
  - Don’t use actual code
  - Instead, use the ideas *behind* code
  - Design patterns document ideas
  - Lots of “tried and tested” solutions

Pattern links

http://www.cetus-links.org/oo_patterns.html
http://www.hillside.net
Design patterns

Object-oriented software engineering has developed a rich language for control models

**Design patterns** = abstract form of re-use:

- Don’t use actual code
- Instead, use the ideas *behind* code
- Design patterns document ideas
- Lots of “tried and tested” solutions

Pattern links

http://www.cetus-links.org/oo_patterns.html
http://www.hillside.net
Design patterns

- Object-oriented software engineering has developed a rich language for control models
- **Design patterns** = abstract form of re-use:
  - Don’t use actual code
  - Instead, use the ideas *behind* code
  - Design patterns document ideas
  - Lots of “tried and tested” solutions

Pattern links

http://www.cetus-links.org/oo_patterns.html
http://www.hillside.net
Design patterns

- Object-oriented software engineering has developed a rich language for control models

- **Design patterns** = abstract form of re-use:
  - Don’t use actual code
  - Instead, use the ideas *behind* code
  - Design patterns document ideas
  - Lots of “tried and tested” solutions

Pattern links

http://www.cetus-links.org/oo_patterns.html
http://www.hillside.net
Development of Design patterns

Recipes for software success? ...no, more abstract but tried+tested solutions – form of re-use

- Origins in architecture: *Christopher Alexander* – *pattern languages for “habitable spaces”*.

- Evolutionary approach to SE observes:
  - Hard to get good solutions first time
  - But experience $\leadsto$ re-use of past successes
  - Consequently, solution is refined and made more abstract
  - Disseminated, shared, published in OO community

Identifying the right pattern to use $\leadsto$ better/faster design
Development of Design patterns

Recipes for software success? ...no, more abstract but tried+tested solutions – form of re-use

- Origins in architecture: Christopher Alexander – pattern languages for “habitable spaces”.
- Evolutionary approach to SE observes:
  - Hard to get good solutions first time
  - But experience $\leadsto$ re-use of past successes
  - Consequently, solution is refined and made more abstract
  - Disseminated, shared, published in OO community

Identifying the right pattern to use $\leadsto$ better/faster design
Development of Design patterns

Recipes for software success? ...no, more abstract but tried+tested solutions – form of re-use

- Origins in architecture: *Christopher Alexander – pattern languages for “habitable spaces”.*
- Evolutionary approach to SE observes:
  - Hard to get good solutions first time
  - But experience $\Rightarrow$ re-use of past successes
  - Consequently, solution is refined and made more abstract
  - Disseminated, shared, published in OO community

Identifying the right pattern to use $\Rightarrow$ better/faster design
Development of Design patterns

Recipes for software success? ...no, more abstract but tried+tested solutions – form of re-use

- Origins in architecture: *Christopher Alexander* – *pattern languages for “habitable spaces”.*

- Evolutionary approach to SE observes:
  - Hard to get good solutions first time
  - But experience \(\mapsto\) re-use of past successes
  - Consequently, solution is refined and made more abstract
  - Disseminated, shared, published in OO community

Identifying the right pattern to use \(\mapsto\) better/faster design
But what are they?

Alexander: “Each pattern describes a problem which occurs over and over again...describes the core of a solution...in such a way that you can use this solution a million times over without ever doing it the same way twice.”

A pattern has four components:

- **Pattern name**: a vocabulary of patterns
- **The problem**: when a pattern is useful – symptoms, conditions
- **The solution**: elements of the design – relationships, responsibilities, collaborations – but not a concrete design
- **The consequences**: results and trade-offs, e.g. impact on flexibility, extensibility, portability
But what are they?

Alexander: “Each pattern describes a problem which occurs over and over again...describes the core of a solution...in such a way that you can use this solution a million times over without ever doing it the same way twice.”

A pattern has four components:

- **Pattern name:** a vocabulary of patterns
- **The problem:** when a pattern is useful – symptoms, conditions
- **The solution:** elements of the design – relationships, responsibilities, collaborations – but not a concrete design
- **The consequences:** results and trade-offs, e.g. impact on flexibility, extensibility, portability
Selected Patterns

- **Abstract factory** Create families of related objects without specifying concrete classes
- **Adapter** Convert interface of a class to make it compatible with a new range of clients
- **Decorator** Dynamic addition of responsibilities to an object – flexible alternative to subclassing
- **Factory method** Interface for creation, but subclasses decide which class to instantiate
- **Observer** Defines one-to-many dependency between objects so that when one object changes state, all dependents are notified

Approx. 20 core patterns, see: [Gamma et al., 1995] and [Freeman et al., 2004]
Model/View/Controller (MVC)

MODEL = the application
VIEW = the screen presentation
CONTROLLER = how system reacts to user input

- MVC *decouples* views and models
- Solid = direct association
- Dashed = indirect association

- Interface: notify protocol
- Views must reflect state of model
- Views derive from Observer pattern
- Complex views constructed with Composite pattern
- Controller hierarchies derive from Strategy pattern
Model/View/Controller (MVC)

MODEL = the application
VIEW = the screen presentation
CONTROLLER = how system reacts to user input

- MVC decouples views and models
- Solid = direct association
- Dashed = indirect association

- interface: notify protocol
- view: must reflect state of model
- Views derive from Observer pattern
- Complex views constructed with Composite pattern
- Controller hierarchies derive from Strategy pattern
Model/View/Controller (MVC)

MODEL = the application
VIEW = the screen presentation
CONTROLLER = how system reacts to user input

- MVC *decouples* views and models
- Solid = direct association
- Dashed = indirect association

Diagram:

- Interface: notify protocol
- View: must reflect state of model
- Views derive from Observer pattern
- Complex views constructed with Composite pattern
- Controller hierarchies derive from Strategy pattern
Model/View/Controller (MVC)

- MODEL = the application
- VIEW = the screen presentation
- CONTROLLER = how system reacts to user input

- MVC *decouples* views and models
- Solid = direct association
- Dashed = indirect association

- interface: notify protocol
- view: must reflect state of model
- Views derive from **Observer** pattern
- Complex views constructed with **Composite** pattern
- Controller hierarchies derive from **Strategy** pattern
Client-server

- Commonest architecture in IT?
  - Client: requests services
  - Server: provides services
  - Variation stems from number of tiers:
    - 2-tier—application logic in either or both tiers:
      1. User interface (client)
      2. Database (server)
    - 3-tier—application logic in between:
      1. User interface (client)
      2. Application logic (server middleware)
      3. Database (server)
    - 3-tier ≠ MVC:
      - 3-tier is linear: presentation and data tier do not communicate directly
      - MVC is triangular: see earlier diagram
      - Origins: 3-tier in distributed applications, MVC in workstation applications
Client-server

- Commonest architecture in IT?
- Client: requests services
  - Server: provides services
- Variation stems from number of tiers:
  - 2-tier—application logic in either or both tiers:
    1. User interface (client)
    2. Database (server)
  - 3-tier—application logic in between:
    1. User interface (client)
    2. Application logic (server middleware)
    3. Database (server)
  - 3-tier ≠ MVC:
    - 3-tier is linear: presentation and data tier do not communicate directly
    - MVC is triangular: see earlier diagram
    - Origins: 3-tier in distributed applications, MVC in workstation applications
Client-server

- Commonest architecture in IT?
- Client: requests services
- Server: provides services

Variation stems from number of tiers:
- 2-tier—application logic in either or both tiers:
  1. User interface (client)
  2. Database (server)
- 3-tier—application logic in between:
  1. User interface (client)
  2. Application logic (server middleware)
  3. Database (server)
- 3-tier ≠ MVC:
  - 3-tier is linear: presentation and data tier do not communicate directly
  - MVC is triangular: see earlier diagram
  - Origins: 3-tier in distributed applications, MVC in workstation applications
Client-server

- Commonest architecture in IT?
- Client: requests services
- Server: provides services
- Variation stems from number of tiers:
  - 2-tier—application logic in either or both tiers:
    1. User interface (client)
    2. Database (server)
  - 3-tier—application logic in between:
    1. User interface (client)
    2. Application logic (server middleware)
    3. Database (server)
  - 3-tier ≠ MVC:
    - 3-tier is linear: presentation and data tier do not communicate directly
    - MVC is triangular: see earlier diagram
    - Origins: 3-tier in distributed applications, MVC in workstation applications
Rich Internet Applications

■ Aim: to offer desktop functionality through a browser
■ Characteristics:
  ■ No need for software installation
  ■ Security through sandboxing (like Java)
  ■ Stateful through service layer on back-end
■ For example: a browser-based chat room, calendar, google docs
Rich Internet Applications

- Aim: to offer desktop functionality through a browser
- Characteristics:
  - No need for software installation
  - Security through sandboxing (like Java)
  - Stateful through service layer on back-end
- For example: a browser-based chat room, calendar, google docs
Rich Internet Applications

- Aim: to offer desktop functionality through a browser
- Characteristics:
  - No need for software installation
  - Security through sandboxing (like Java)
  - Stateful through service layer on back-end
- For example: a browser-based chat room, calendar, google docs
Motivations

- Origins:
  - Macromedia (2002)
  - Remote scripting (ca. 1998) — Microsoft, early version of XMLHttpRequest etc.
  - Rich (web) clients/applications

- Contrast with: client-server/thin-client style applications

- Division of labour: server does work, client renders content

- Driver: responsiveness intrinsically difficult to achieve

- Solution: client-side software to avoid slow loop to (web) server

- Distinguishing characteristic: a layer of code executing in the browser:
  - Takes over rendering task
  - Takes over server communications
Motivations

Origins:
- Macromedia (2002)
- Remote scripting (ca. 1998) — Microsoft, early version of XMLHttpRequest etc.
- Rich (web) clients/applications

Contrast with: client-server/thin-client style applications
- Division of labour: server does work, client renders content
- Driver: responsiveness intrinsically difficult to achieve
- Solution: client-side software to avoid slow loop to (web) server
- Distinguishing characteristic: a layer of code executing in the browser:
  - Takes over rendering task
  - Takes over server communications
Motivations

- Origins:
  - Macromedia (2002)
  - Remote scripting (ca. 1998) — Microsoft, early version of XMLHttpRequest etc.
  - Rich (web) clients/applications

- Contrast with: client-server/thin-client style applications

- Division of labour: server does work, client renders content

- Driver: responsiveness intrinsically difficult to achieve

- Solution: client-side software to avoid slow loop to (web) server

- Distinguishing characteristic: a layer of code executing in the browser:
  - Takes over rendering task
  - Takes over server communications
Motivations

- **Origins:**
  - Macromedia (2002)
  - Remote scripting (ca. 1998) — Microsoft, early version of XMLHttpRequest etc.
  - Rich (web) clients/applications

- **Contrast with:** client-server/thin-client style applications

- **Division of labour:** server does work, client renders content

- **Driver:** responsiveness intrinsically difficult to achieve

  - **Solution:** client-side software to avoid slow loop to (web) server

- **Distinguishing characteristic:** a layer of code executing in the browser:
  - Takes over rendering task
  - Takes over server communications
Motivations

- Origins:
  - Macromedia (2002)
  - Remote scripting (ca. 1998) — Microsoft, early version of XMLHttpRequest etc.
  - Rich (web) clients/applications

- Contrast with: client-server/thin-client style applications

- Division of labour: server does work, client renders content

- Driver: responsiveness intrinsically difficult to achieve

- Solution: client-side software to avoid slow loop to (web) server

- Distinguishing characteristic: a layer of code executing in the browser:
  - Takes over rendering task
  - Takes over server communications
Motivations

- Origins:
  - Macromedia (2002)
  - Remote scripting (ca. 1998) — Microsoft, early version of XMLHttpRequest etc.
  - Rich (web) clients/applications

- Contrast with: client-server/thin-client style applications

- Division of labour: server does work, client renders content

- Driver: responsiveness intrinsically difficult to achieve

- Solution: client-side software to avoid slow loop to (web) server

- Distinguishing characteristic: a layer of code executing in the browser:
  - Takes over rendering task
  - Takes over server communications
Benefits

- Much easier to maintain up-to-date installations
- No overhead for update or distribution
- Users can use application on any computer anywhere
- Not necessary to be always on-line
- Cross-platform consistency
- Reduced risk of viruses compared to conventional applications

Tools: Adobe AIR, Google Gears, Curl
Benefits

- Much easier to maintain up-to-date installations
- No overhead for update or distribution
- Users can use application on any computer anywhere
- Not necessary to be always online
- Cross-platform consistency
- Reduced risk of viruses compared to conventional applications

Tools: Adobe AIR, Google Gears, Curl
Benefits

- Much easier to maintain up-to-date installations
- No overhead for update or distribution
- Users can use application on any computer anywhere
  - Not necessary to be always on-line
  - Cross-platform consistency
  - Reduced risk of viruses compared to conventional applications

Tools: Adobe AIR, Google Gears, Curl
Benefits

- Much easier to maintain up-to-date installations
- No overhead for update or distribution
- Users can use application on any computer anywhere
- Not necessary to be always on-line
- Cross-platform consistency
- Reduced risk of viruses compared to conventional applications

Tools: Adobe AIR, Google Gears, Curl
Benefits

- Much easier to maintain up-to-date installations
- No overhead for update or distribution
- Users can use application on any computer anywhere
- Not necessary to be always on-line
- Cross-platform consistency
- Reduced risk of viruses compared to conventional applications

Tools: Adobe AIR, Google Gears, Curl
Benefits

- Much easier to maintain up-to-date installations
- No overhead for update or distribution
- Users can use application on any computer anywhere
- Not necessary to be always on-line
- Cross-platform consistency
- Reduced risk of viruses compared to conventional applications

Tools: Adobe AIR, Google Gears, Curl
Comparison

- “richer”: because client-side functionality not restricted to what you can do in HTML
- “more responsive”: because many operations can be processed client-side
- Changes client-server balance: interaction computational load moved to client
- Asynchronous communication:
  - client-side can interact with server when necessary; not on each button click
  - client-side can pre-fetch data—pros + cons
  - potential increased network efficiency: derived from smarter client-server interaction
Comparison

- “richer”: because client-side functionality not restricted to what you can do in HTML
- “more responsive”: because many operations can be processed client-side

Changes client-server balance: interaction computational load moved to client

Asynchronous communication:
  - client-side can interact with server when necessary; not on each button click
  - client-side can pre-fetch data—pros + cons
  - potential increased network efficiency: derived from smarter client-server interaction
Comparison

- “richer”: because client-side functionality not restricted to what you can do in HTML
- “more responsive”: because many operations can be processed client-side
- Changes client-server balance: interaction computational load moved to client

Asynchronous communication:

- client-side can interact with server when necessary; not on each button click
- client-side can pre-fetch data—pros + cons
- potential increased network efficiency: derived from smarter client-server interaction
Comparison

- “richer”: because client-side functionality not restricted to what you can do in HTML
- “more responsive”: because many operations can be processed client-side
- Changes client-server balance: interaction computational load moved to client
- Asynchronous communication:
  - client-side can interact with server when necessary; not on each button click
  - client-side can pre-fetch data—pros + cons
  - potential increased network efficiency: derived from smarter client-server interaction
Drawbacks I

- **Sandbox:** means application is operating in a restricted environment
- **Scripting:** many applications use {Java/Ecma}script—security + performance issues
- **Performance:** compiled client-side $\mapsto$ Java, Curl, Silverlight, Flash or Javascript JiT compilers
- **Script download:** an overhead... compress and stage
- **XHTML tension:** purpose of RIA is to be in charge of rendering etc., but principle of XHTML (as delivered from server) is to let browser decide
- **Non-indexable:** dynamic pages not visible
- **Always-on:** vs. sometimes off-line
- **Accessibility:** dynamic pages do not work well with screen readers
Drawbacks II

- **Security**: local access varies between RIA frameworks—makes for portability and security challenges... increased risk of cross-site scripting
Status I

- Dependence on latest version of browsers for performance
- Classical (!) problem of browser compliance
- Development environments emerging:
  - Ajax — discussed
  - Flash + Flex (Adobe, formerly Macromedia)
    - Flex data services: IDE (Eclipse) + J2EE
    - Actionscript ≈ ECMAscript
    - Presentation tier
  - Curl
    - Reflective OOPL developed at MIT
    - Markup language + scripting supported by JiT plugin
  - Microsoft Silverlight (Moonlight)
    - Plugin for client-side media processing
    - Markup with Extensible Application Markup Language (XAML), scripting with Javascript
  - Google Web Toolkit (GWT)
    - Java SDK for Ajax
Status II

- Cross-compiles Java to Javascript for client-side
- ... and lots more
- W3C Rich Web Clients: http://www.w3.org/2006/rwc/
Content

1. Agile Development
2. Design Patterns
3. Web Architectures
4. RIAs
5. Web Services
   - WSDL
   - SOAP
   - UDDI
   - REST
   - Workflows
6. Summary
What is a web service?

- A software component
  - Stored on one computer
  - Accessible via method calls by an application
  - From another computer over a network

- Web services typically communicate using XML and HTTP
What is a web service?

- A software component
  - Stored on one computer
  - Accessible via method calls by an application
  - From another computer over a network
- Web services typically communicate using XML and HTTP
Web Services: Why?

- Business model: pay-per-use
- Retention of control
- Reduction of maintenance costs
- Low distribution costs
- Focus on function not deployment
- Ease access to large computational resources
- New method of abstraction
- Transition from “web for humans” to “web for programs”
Web Services: Why?

- Business model: pay-per-use
- Retention of control
- Reduction of maintenance costs
- Low distribution costs
- Focus on function not deployment
- Ease access to large computational resources
- New method of abstraction
- Transition from “web for humans” to “web for programs”
Web Services: Why?

- Business model: pay-per-use
- Retention of control
- Reduction of maintenance costs
  - Low distribution costs
  - Focus on function not deployment
  - Ease access to large computational resources
  - New method of abstraction
  - Transition from “web for humans” to “web for programs”
Web Services: Why?

- Business model: pay-per-use
- Retention of control
- Reduction of maintenance costs
- Low distribution costs
  - Focus on function not deployment
  - Ease access to large computational resources
  - New method of abstraction
  - Transition from “web for humans” to “web for programs”
Web Services: Why?

- Business model: pay-per-use
- Retention of control
- Reduction of maintenance costs
- Low distribution costs
- Focus on function not deployment
- Ease access to large computational resources
- New method of abstraction
- Transition from “web for humans” to “web for programs”
Web Services: Why?

- Business model: pay-per-use
- Retention of control
- Reduction of maintenance costs
- Low distribution costs
- Focus on function not deployment
- Ease access to large computational resources
- New method of abstraction
- Transition from “web for humans” to “web for programs”
Web Services: Why?

- Business model: pay-per-use
- Retention of control
- Reduction of maintenance costs
- Low distribution costs
- Focus on function not deployment
- Ease access to large computational resources
- New method of abstraction

- Transition from “web for humans” to “web for programs”
Web Services: Why?

- Business model: pay-per-use
- Retention of control
- Reduction of maintenance costs
- Low distribution costs
- Focus on function not deployment
- Ease access to large computational resources
- New method of abstraction
- Transition from “web for humans” to “web for programs”
Web Services: How?

- Web “objects”: SOAP or REST
- WS-Agreement
- Service level agreements (SLAs)
- Workflow specification languages:
  - Business Process Execution Language (BPEL)
  - Yet Another Workflow Language (YAWL)
- Workflow enactment engines:
  - Science: Taverna, Triana, Kepler, Gridbus
  - Engineering: ANSYS, Noesis, Engineous, Phoenix
  - Business: Apache ODE, JBPM (JBoss)
Web Services: How?

- Web “objects”: SOAP or REST
- WS-Agreement
  - Service level agreements (SLAs)
  - Workflow specification languages:
    - Business Process Execution Language (BPEL)
    - Yet Another Workflow Language (YAWL)
  - Workflow enactment engines:
    - Science: Taverna, Triana, Kepler, Gridbus
    - Engineering: ANSYS, Noesis, Engineous, Phoenix
    - Business: Apache ODE, JBPMM (JBoss)
Web Services: How?

- Web “objects”: SOAP or REST
- WS-Agreement
- Service level agreements (SLAs)
  - Workflow specification languages:
    - Business Process Execution Language (BPEL)
    - Yet Another Workflow Language (YAWL)
  - Workflow enactment engines:
    - Science: Taverna, Triana, Kepler, Gridbus
    - Engineering: ANSYS, Noesis, Engineous, Phoenix
    - Business: Apache ODE, JBPMS (JBoss)
Web Services: How?

- Web “objects”: SOAP or REST
- WS-Agreement
- Service level agreements (SLAs)
- Workflow specification languages:
  - Business Process Execution Language (BPEL)
  - Yet Another Workflow Language (YAWL)
- Workflow enactment engines:
  - Science: Taverna, Triana, Kepler, Gridbus
  - Engineering: ANSYS, Noesis, Engineous, Phoenix
  - Business: Apache ODE, JBPM (JBoss)
Web Services: How?

- Web “objects”: SOAP or REST
- WS-Agreement
- Service level agreements (SLAs)
- Workflow specification languages:
  - Business Process Execution Language (BPEL)
  - Yet Another Workflow Language (YAWL)
- Workflow enactment engines:
  - Science: Taverna, Triana, Kepler, Gridbus
  - Engineering: ANSYS, Noesis, Engineous, Phoenix
  - Business: Apache ODE, JBPM (JBoss)
Web Services: Challenges?

1. Service description: interface? function?
2. Service advertisement: where?
3. Service discovery: matching + brokerage
4. Service contracts: governance
5. Service composition: workflow definition and re-use
6. Orchestration vs. Choreography
7. Provenance: automation of audit trails
Web Services: Challenges?

1. Service description: interface? function?
2. Service advertisement: where?
3. Service discovery: matching + brokerage
4. Service contracts: governance
5. Service composition: workflow definition and re-use
6. Orchestration vs. Choreography
7. Provenance: automation of audit trails
Web Services: Challenges?

1. Service description: interface? function?
2. Service advertisement: where?
3. Service discovery: matching + brokerage
4. Service contracts: governance
5. Service composition: workflow definition and re-use
6. Orchestration vs. Choreography
7. Provenance: automation of audit trails
Web Services: Challenges?

1. Service description: interface? function?
2. Service advertisement: where?
3. Service discovery: matching + brokerage
4. Service contracts: governance
5. Service composition: workflow definition and re-use
6. Orchestration vs. Choreography
7. Provenance: automation of audit trails
Web Services: Challenges?

1. Service description: interface? function?
2. Service advertisement: where?
3. Service discovery: matching + brokerage
4. Service contracts: governance
5. Service composition: workflow definition and re-use
6. Orchestration vs. Choreography
7. Provenance: automation of audit trails
Web Services: Challenges?

1. Service description: interface? function?
2. Service advertisement: where?
3. Service discovery: matching + brokerage
4. Service contracts: governance
5. Service composition: workflow definition and re-use
6. Orchestration vs. Choreography
7. Provenance: automation of audit trails
Web Services: Challenges?

1. Service description: interface? function?
2. Service advertisement: where?
3. Service discovery: matching + brokerage
4. Service contracts: governance
5. Service composition: workflow definition and re-use
6. Orchestration vs. Choreography
7. Provenance: automation of audit trails
WSDL

- To use a web service
  - Must know where to find the web service
  - Must have the web service’s description

- Web Service Description Language (WSDL):
  - Describe web services in a platform-independent manner
  - Server generates a web service’s WSDL dynamically
  - Client tools parse the WSDL to create the client-side proxy class that accesses the web service

- Current version 2.0; version 1.1 not endorsed by the W3C but version 2.0 is a W3C recommendation.

- WSDL 1.2 renamed WSDL 2.0 because of substantial differences

- Adds binding to all the HTTP request methods (not just GET and POST)

- Hence better support for RESTful web services
WSDL

- To use a web service
  - Must know where to find the web service
  - Must have the web service’s description

- Web Service Description Language (WSDL):
  - Describe web services in a platform-independent manner
  - Server generates a web service’s WSDL dynamically
  - Client tools parse the WSDL to create the client-side proxy class that accesses the web service

- Current version 2.0; version 1.1 not endorsed by the W3C but version 2.0 is a W3C recommendation.

- WSDL 1.2 renamed WSDL 2.0 because of substantial differences

- Adds binding to all the HTTP request methods (not just GET and POST)

- Hence better support for RESTful web services
WSDL

To use a web service

- Must know where to find the web service
- Must have the web service's description

Web Service Description Language (WSDL):

- Describe web services in a platform-independent manner
- Server generates a web service's WSDL dynamically
- Client tools parse the WSDL to create the client-side proxy class that accesses the web service

Current version 2.0; version 1.1 not endorsed by the W3C but version 2.0 is a W3C recommendation.

- WSDL 1.2 renamed WSDL 2.0 because of substantial differences
- Adds binding to all the HTTP request methods (not just GET and POST)
- Hence better support for RESTful web services

WSDL

To use a web service

- Must know where to find the web service
- Must have the web service's description

Web Service Description Language (WSDL):

- Describe web services in a platform-independent manner
- Server generates a web service's WSDL dynamically
- Client tools parse the WSDL to create the client-side proxy class that accesses the web service

Current version 2.0; version 1.1 not endorsed by the W3C but version 2.0 is a W3C recommendation.

- WSDL 1.2 renamed WSDL 2.0 because of substantial differences
- Adds binding to all the HTTP request methods (not just GET and POST)
- Hence better support for RESTful web services
WSDL

- To use a web service
  - Must know where to find the web service
  - Must have the web service’s description

- Web Service Description Language (WSDL):
  - Describe web services in a platform-independent manner
  - Server generates a web service’s WSDL dynamically
  - Client tools parse the WSDL to create the client-side proxy class that accesses the web service

- Current version 2.0; version 1.1 not endorsed by the W3C but version 2.0 is a W3C recommendation.

- WSDL 1.2 renamed WSDL 2.0 because of substantial differences
  - Adds binding to all the HTTP request methods (not just GET and POST)
  - Hence better support for RESTful web services
WSDL

- To use a web service
  - Must know where to find the web service
  - Must have the web service’s description

- Web Service Description Language (WSDL):
  - Describe web services in a platform-independent manner
  - Server generates a web service’s WSDL dynamically
  - Client tools parse the WSDL to create the client-side proxy class that accesses the web service

- Current version 2.0; version 1.1 not endorsed by the W3C but version 2.0 is a W3C recommendation.

- WSDL 1.2 renamed WSDL 2.0 because of substantial differences

- Adds binding to all the HTTP request methods (not just GET and POST)
  - Hence better support for RESTful web services
WSDL

■ To use a web service
  ■ Must know where to find the web service
  ■ Must have the web service’s description

■ Web Service Description Language (WSDL):
  ■ Describe web services in a platform-independent manner
  ■ Server generates a web service’s WSDL dynamically
  ■ Client tools parse the WSDL to create the client-side proxy class that accesses the web service

■ Current version 2.0; version 1.1 not endorsed by the W3C but version 2.0 is a W3C recommendation.

■ WSDL 1.2 renamed WSDL 2.0 because of substantial differences

■ Adds binding to all the HTTP request methods (not just GET and POST)

■ Hence better support for RESTful web services
WSDL 2.0 Elements

- Abstract definition
  - Message
    - Used to communicate with the WS
    - Typed definitions of data being exchanged
  - Interface
    - Abstract definition of a service
    - A group of operations offered by one endpoint of the WS: An operation is an abstract description of an action and refers to input and/or output messages
    - There can be more than one interface for a single WS

- Concrete
  - Binding
    - Maps an interface to a concrete protocol and data format (e.g. SOAP1.1 over HTTP)
  - Service
    - Aggregate set of related endpoints
    - Maps each binding to an endpoint (network address: URL for HTTP)
WSDL 2.0 Elements

- Abstract definition
  - Message
    - Used to communicate with the WS
    - Typed definitions of data being exchanged
  - Interface
    - Abstract definition of a service
    - A group of operations offered by one endpoint of the WS: An operation is an abstract description of an action and refers to input and/or output messages
    - There can be more than one interface for a single WS

- Concrete
  - Binding
    - Maps an interface to a concrete protocol and data format (e.g. SOAP1.1 over HTTP)
  - Service
    - Aggregate set of related endpoints
    - Maps each binding to an endpoint (network address: URL for HTTP)
portType and operations

- A portType is a named set of operations offered by the Web service
- An operation itself is a name associated with a particular exchange of messages
- Each operation consists of a pattern of messages
  - Message names must be namespace-qualified
  - The sequence of messages defines the behavior of the operation
  - Four basic patterns:
    1. Request-response: Input then Output
    2. Solicit-response: Output then Input
    3. Notification: Output only
    4. One-way: Input only
portType and operations

- A portType is a named set of operations offered by the Web service.
- An operation itself is a name associated with a particular exchange of messages.
- Each operation consists of a pattern of messages:
  - Message names must be namespace-qualified.
  - The sequence of messages defines the behavior of the operation.
  - Four basic patterns:
    1. Request-response: Input then Output
    2. Solicit-response: Output then Input
    3. Notification: Output only
    4. One-way: Input only
portType and operations

- A portType is a named set of operations offered by the Web service.
- An operation itself is a name associated with a particular exchange of messages.
- Each operation consists of a pattern of messages:
  - Message names must be namespace-qualified.
  - The sequence of messages defines the behavior of the operation.
  - Four basic patterns:
    1. Request-response: Input then Output
    2. Solicit-response: Output then Input
    3. Notification: Output only
    4. One-way: Input only
WSDL Example

```xml
<wSDL:message name="GetBookPriceInput">
    GetBookPriceInput
    <wSDL:part name="body" element="xsd1:BookRequest"/>
</wSDL:message>
<wSDL:message name="GetBookPriceOutput">
    GetBookPriceOutput
    <!-- Zero or more part elements -->
    <wSDL:part name="body" element="xsd1:BookPrice"/>
</wSDL:message>
<wSDL:portType name=BookStorePortType">
    BookStorePortType
    <!-- Combines multiple messages to form operations -->
    <wSDL:operation name="GetBookPrice">
        GetBookPrice
        <wSDL:input message="tns:GetBookPriceInput"/>
        <wSDL:output message="tns:GetBookPriceOutput"/>
    </wSDL:operation>
    <!-- More operations -->
</wSDL:portType>
```
WSDL-SOAP Binding

- A binding defines message format and protocol details for operations and messages defined by a particular portType
- Built-in extensions for SOAP-specific details
- `<soap:binding>`
  - Indicates binding will be made available via SOAP
  - Style attribute indicates message format:
    - document, simple XML documents
    - rpc, additional wrapper element indicating the function name
- `<soap:operation>`
  - Indicates binding of a specific operation to a specific SOAP implementation
- `<soap:body>`
  - For each operation, specifies details of the input/output messages, such as encoding, header blocks, fault, ...
- `<soap:address>`
  - location: where service is accessible
SOAP

- Originally conceived to bridge the gap between disparate RPC-based communication platforms
- SOAP (was) acronym: Simple Object Access Protocol
- Evolved into the most widely supported protocol for use with XML web services
- Establishes a standard message format; an XML document capable of hosting RPC but also document-centric data
- SOAP is not a particularly good solution
  - Inefficient due to character (not binary) data and large headers
  - Unlikely to replace other distributed computing technologies (e.g. RMI)
- Developed by Microsoft, IBM, DevelopMentor, UserLand and Lotus
- SOAP 1.1 specifications: industrial standard
- SOAP 1.2 specifications: W3C Recommendation
SOAP

- Originally conceived to bridge the gap between disparate RPC-based communication platforms
- SOAP (was) acronym: Simple Object Access Protocol
  - Evolved into the most widely supported protocol for use with XML web services
  - Establishes a standard message format; an XML document capable of hosting RPC but also document-centric data
  - SOAP is not a particularly good solution
    - Inefficient due to character (not binary) data and large headers
    - Unlikely to replace other distributed computing technologies (e.g. RMI)
- Developed by Microsoft, IBM, DevelopMentor, UserLand and Lotus
- SOAP 1.1 specifications: industrial standard
- SOAP 1.2 specifications: W3C Recommendation
SOAP

- Originally conceived to bridge the gap between disparate RPC-based communication platforms
- SOAP (was) acronym: Simple Object Access Protocol
- Evolved into the most widely supported protocol for use with XML web services

- Establishes a standard message format; an XML document capable of hosting RPC but also document-centric data
- SOAP is not a particularly good solution
  - Inefficient due to character (not binary) data and large headers
  - Unlikely to replace other distributed computing technologies (e.g. RMI)
- Developed by Microsoft, IBM, DevelopMentor, UserLand and Lotus
- SOAP 1.1 specifications: industrial standard
- SOAP 1.2 specifications: W3C Recommendation
SOAP

- Originally conceived to bridge the gap between disparate RPC-based communication platforms
- SOAP (was) acronym: Simple Object Access Protocol
- Evolved into the most widely supported protocol for use with XML web services
- Establishes a standard message format; an XML document capable of hosting RPC but also document-centric data
- SOAP is not a particularly good solution
  - Inefficient due to character (not binary) data and large headers
  - Unlikely to replace other distributed computing technologies (e.g. RMI)
- Developed by Microsoft, IBM, DevelopMentor, UserLand and Lotus
- SOAP 1.1 specifications: industrial standard
- SOAP 1.2 specifications: W3C Recommendation
SOAP

- Originally conceived to bridge the gap between disparate RPC-based communication platforms
- SOAP (was) acronym: Simple Object Access Protocol
- Evolved into the most widely supported protocol for use with XML web services
- Establishes a standard message format; an XML document capable of hosting RPC but also document-centric data
- SOAP is not a particularly good solution
  - Inefficient due to character (not binary) data and large headers
  - Unlikely to replace other distributed computing technologies (e.g. RMI)
- Developed by Microsoft, IBM, DevelopMentor, UserLand and Lotus
- SOAP 1.1 specifications: industrial standard
- SOAP 1.2 specifications: W3C Recommendation

Originally conceived to bridge the gap between disparate RPC-based communication platforms

SOAP (was) acronym: Simple Object Access Protocol

Evolved into the most widely supported protocol for use with XML web services

Establishes a standard message format; an XML document capable of hosting RPC but also document-centric data

SOAP is not a particularly good solution

- Inefficient due to character (not binary) data and large headers
- Unlikely to replace other distributed computing technologies (e.g. RMI)

Developed by Microsoft, IBM, DevelopMentor, UserLand and Lotus

SOAP 1.1 specifications: industrial standard

SOAP 1.2 specifications: W3C Recommendation
SOAP

- Originally conceived to bridge the gap between disparate RPC-based communication platforms
- SOAP (was) acronym: Simple Object Access Protocol
- Evolved into the most widely supported protocol for use with XML web services
- Establishes a standard message format; an XML document capable of hosting RPC but also document-centric data
- SOAP is not a particularly good solution
  - Inefficient due to character (not binary) data and large headers
  - Unlikely to replace other distributed computing technologies (e.g. RMI)
- Developed by Microsoft, IBM, DevelopMentor, UserLand and Lotus

- SOAP 1.1 specifications: industrial standard
- SOAP 1.2 specifications: W3C Recommendation
SOAP

- Originally conceived to bridge the gap between disparate RPC-based communication platforms
- SOAP (was) acronym: Simple Object Access Protocol
- Evolved into the most widely supported protocol for use with XML web services
- Establishes a standard message format; an XML document capable of hosting RPC but also document-centric data
- SOAP is not a particularly good solution
  - Inefficient due to character (not binary) data and large headers
  - Unlikely to replace other distributed computing technologies (e.g. RMI)
- Developed by Microsoft, IBM, DevelopMentor, UserLand and Lotus
- SOAP 1.1 specifications: industrial standard
- SOAP 1.2 specifications: W3C Recommendation
SOAP

- Originally conceived to bridge the gap between disparate RPC-based communication platforms
- SOAP (was) acronym: Simple Object Access Protocol
- Evolved into the most widely supported protocol for use with XML web services
- Establishes a standard message format; an XML document capable of hosting RPC but also document-centric data
- SOAP is not a particularly good solution
  - Inefficient due to character (not binary) data and large headers
  - Unlikely to replace other distributed computing technologies (e.g. RMI)
- Developed by Microsoft, IBM, DevelopMentor, UserLand and Lotus
- SOAP 1.1 specifications: industrial standard
- SOAP 1.2 specifications: W3C Recommendation
SOAP elements

- **Message envelope**: format for messages + extensions
  - Encoding rules:
    - Rules for encoding common data types and application-defined data types in XML form
    - Messages are constructed using the data types defined in W3C XML schema
  - RPC convention
  - Asynchronous (one-way) messages
  - Binding with underlying protocols, e.g., over HTTP
  - SOAP with attachments
    - MIME used to define message body format to allow multi-part textual and non-textual (non-ASCII) message bodies; in particular:
      - Multiple objects in a single message
      - Representation of body text in various character sets
      - Representation of non-textual material: images, audio fragments, programs, ..., and in general, binary files
SOAP elements

- Message envelope: format for messages + extensions
- Encoding rules:
  - Rules for encoding common data types and application-defined data types in XML form
  - Messages are constructed using the data types defined in W3C XML schema
- RPC convention
- Asynchronous (one-way) messages
- Binding with underlying protocols, e.g. over HTTP
- SOAP with attachments
  - MIME used to define message body format to allow multi-part textual and non-textual (non-ASCII) message bodies; in particular:
    - Multiple objects in a single message
    - Representation of body text in various character sets
    - Representation of non-textual material: images, audio fragments, programs, ..., and in general, binary files
SOAP elements

- Message envelope: format for messages + extensions

- Encoding rules:
  - Rules for encoding common data types and application-defined data types in XML form
  - Messages are constructed using the data types defined in W3C XML schema

- RPC convention
  - Asynchronous (one-way) messages
  - Binding with underlying protocols, e.g. over HTTP
  - SOAP with attachments
    - MIME used to defines message body format to allow multi-part textual and non-textual (non-ASCII) message bodies; in particular:
      - Multiple objects in a single message
      - Representation of body text in various character sets
      - Representation of non-textual material: images, audio fragments, programs, ..., and in general, binary files
SOAP elements

- Message envelope: format for messages + extensions
- Encoding rules:
  - Rules for encoding common data types and application-defined data types in XML form
  - Messages are constructed using the data types defined in W3C XML schema
- RPC convention
- Asynchronous (one-way) messages
  - Binding with underlying protocols, e.g. over HTTP
  - SOAP with attachments
    - MIME used to defines message body format to allow multi-part textual and non-textual (non-ASCII) message bodies; in particular:
      - Multiple objects in a single message
      - Representation of body text in various character sets
      - Representation of non-textual material: images, audio fragments, programs, ..., and in general, binary files
SOAP elements

- Message envelope: format for messages + extensions
- Encoding rules:
  - Rules for encoding common data types and application-defined data types in XML form
  - Messages are constructed using the data types defined in W3C XML schema
- RPC convention
- Asynchronous (one-way) messages
- Binding with underlying protocols, e.g. over HTTP
- SOAP with attachments
  - MIME used to defines message body format to allow multi-part textual and non-textual (non-ASCII) message bodies; in particular:
    - Multiple objects in a single message
    - Representation of body text in various character sets
    - Representation of non-textual material: images, audio fragments, programs, ..., and in general, binary files
SOAP elements

- Message envelope: format for messages + extensions
- Encoding rules:
  - Rules for encoding common data types and application-defined data types in XML form
  - Messages are constructed using the data types defined in W3C XML schema
- RPC convention
- Asynchronous (one-way) messages
- Binding with underlying protocols, e.g. over HTTP
- SOAP with attachments
  - MIME used to defines message body format to allow multi-part textual and non-textual (non-ASCII) message bodies; in particular:
    - Multiple objects in a single message
    - Representation of body text in various character sets
    - Representation of non-textual material: images, audio fragments, programs, ..., and in general, binary files
SOAP Message: Envelope

- Envelope is an XML schema
  
  ```xml
  <env:Envelope xmlns:env=http://www.w3.org/2003/05/soap-envelope>
    <env:Header>
      ...
    </env:Header>
    <env:Body>
      <!-- payload -->
      ...
    </env:Body>
  </env:Envelope>
  ```

- Header: SOAP extensions (e.g. ebXML), identification of SOAP intermediaries
- Body: application data, RPC methods + parameters
- More flexible than RPC: separates data from code + any data can be passed
- But application must encode and decode data
- Allows disconnected operation: Queued vs. Direct
- Two message exchange patterns: HTTP POST (request-response) and HTTP GET (response)
XML registry services

- Infrastructure for publishing and discovery of web services

- UDDI = Universal Description, Discovery and Integration

- Two components:
  - Standards-based specifications for service description and discovery
  - Shared operation of a business registry on the web–UDDI registry is itself a web service

- Supported by Microsoft, IBM, HP, Oracle, SAP, Accenture, Ariba, Commerce One

- A “phone directory” for Web services that lists available Web services from different companies, their descriptions and instructions for using them

- A DNS for business applications

- Managed by OASIS standards body (see Resources)
XML registry services

- Infrastructure for publishing and discovery of web services

- **UDDI** = Universal Description, Discovery and Integration

- Two components:
  - Standards-based specifications for service description and discovery
  - Shared operation of a business registry on the web–UDDI registry is itself a web service

- Supported by Microsoft, IBM, HP, Oracle, SAP, Accenture, Ariba, Commerce One

- A “phone directory” for Web services that lists available Web services from different companies, their descriptions and instructions for using them

- A DNS for business applications

- Managed by OASIS standards body (see Resources)
XML registry services

- Infrastructure for publishing and discovery of web services
- UDDI = Universal Description, Discovery and Integration
- Two components:
  - Standards-based specifications for service description and discovery
  - Shared operation of a business registry on the web–UDDI registry is itself a web service
- Supported by Microsoft, IBM, HP, Oracle, SAP, Accenture, Ariba, Commerce One
- A “phone directory” for Web services that lists available Web services from different companies, their descriptions and instructions for using them
- A DNS for business applications
- Managed by OASIS standards body (see Resources)
XML registry services

- Infrastructure for publishing and discovery of web services
- UDDI = Universal Description, Discovery and Integration
- Two components:
  - Standards-based specifications for service description and discovery
  - Shared operation of a business registry on the web–UDDI registry is itself a web service
- Supported by Microsoft, IBM, HP, Oracle, SAP, Accenture, Ariba, Commerce One
  - A “phone directory” for Web services that lists available Web services from different companies, their descriptions and instructions for using them
  - A DNS for business applications
  - Managed by OASIS standards body (see Resources)
XML registry services

- Infrastructure for publishing and discovery of web services
- UDDI = Universal Description, Discovery and Integration
- Two components:
  - Standards-based specifications for service description and discovery
  - Shared operation of a business registry on the web—UDDI registry is itself a web service
- Supported by Microsoft, IBM, HP, Oracle, SAP, Accenture, Ariba, Commerce One
- A “phone directory” for Web services that lists available Web services from different companies, their descriptions and instructions for using them
- A DNS for business applications
- Managed by OASIS standards body (see Resources)
XML registry services

- Infrastructure for publishing and discovery of web services

- UDDI = Universal Description, Discovery and Integration

- Two components:
  - Standards-based specifications for service description and discovery
  - Shared operation of a business registry on the web—UDDI registry is itself a web service

- Supported by Microsoft, IBM, HP, Oracle, SAP, Accenture, Ariba, Commerce One

- A “phone directory” for Web services that lists available Web services from different companies, their descriptions and instructions for using them

- A DNS for business applications

- Managed by OASIS standards body (see Resources)
XML registry services

- Infrastructure for publishing and discovery of web services
- UDDI = Universal Description, Discovery and Integration
- Two components:
  - Standards-based specifications for service description and discovery
  - Shared operation of a business registry on the web–UDDI registry is itself a web service
- Supported by Microsoft, IBM, HP, Oracle, SAP, Accenture, Ariba, Commerce One
- A “phone directory” for Web services that lists available Web services from different companies, their descriptions and instructions for using them
- A DNS for business applications
- Managed by OASIS standards body (see Resources)
UDDI: current status

- Dead!
- Not widely adopted: major companies closed services in 2006
- TC completed 2007, then disbanded
- Microsoft removing it from Windows Server 2010
- Reasons?
UDDI: current status

- Dead!
- Not widely adopted: major companies closed services in 2006
- TC completed 2007, then disbanded
- Microsoft removing it from Windows Server 2010
- Reasons?
UDDI: current status

- Dead!
- Not widely adopted: major companies closed services in 2006
- TC completed 2007, then disbanded
- Microsoft removing it from Windows Server 2010
- Reasons?
UDDI: current status

- Dead!
- Not widely adopted: major companies closed services in 2006
- TC completed 2007, then disbanded
- Microsoft removing it from Windows Server 2010

Reasons?
UDDI: current status

- Dead!
- Not widely adopted: major companies closed services in 2006
- TC completed 2007, then disbanded
- Microsoft removing it from Windows Server 2010
- Reasons?
RESTful web services

- **REST** = REpresentation State Transfer [Fielding, 2000]

Collection of network architecture principles defining:
- resource definition
- resource addressing

Claims to align with key attributes of the Web:

1. Application state and functionality are abstracted into resources
2. Every resource is uniquely addressable using a universal syntax for use in hypermedia links
3. All resources share a uniform interface for the transfer of state between client and resource, consisting of:
   - A constrained set of well-defined operations
   - A constrained set of content types, optionally supporting code on demand
4. A protocol that is: Client-server + Stateless + Cacheable + Layered
RESTful web services

- REST = REpresentation State Transfer [Fielding, 2000]
- Collection of network architecture principles defining:
  - resource definition
  - resource addressing

- Claims to align with key attributes of the Web:
  1. Application state and functionality are abstracted into resources
  2. Every resource is uniquely addressable using a universal syntax for use in hypermedia links
  3. All resources share a uniform interface for the transfer of state between client and resource, consisting of
     - A constrained set of well-defined operations
     - A constrained set of content types, optionally supporting code on demand
  4. A protocol that is: Client-server + Stateless + Cacheable + Layered
RESTful web services

- REST = REpresentation State Transfer [Fielding, 2000]
- Collection of network architecture principles defining:
  - resource definition
  - resource addressing
- Claims to align with key attributes of the Web:
  1. Application state and functionality are abstracted into resources
  2. Every resource is uniquely addressable using a universal syntax for use in hypermedia links
  3. All resources share a uniform interface for the transfer of state between client and resource, consisting of
     - A constrained set of well-defined operations
     - A constrained set of content types, optionally supporting code on demand
  4. A protocol that is: Client-server + Stateless + Cacheable + Layered
Key concept in REST is the “resource” or “source of information”

- Each resource is referenceable via a global identifier (e.g. URI)
- Resource manipulation via standardized interfaces (e.g. HTTP)
- Resource exchange via documents representing the information
- Note: REST was conceived for working with information and media

*Position: The Web is REST, so why not build services that way?*
Resources

- Key concept in REST is the “resource” or “source of information”
- Each resource is referenceable via a global identifier (e.g. URI)
- Resource manipulation via standardized interfaces (e.g. HTTP)
- Resource exchange via documents representing the information
- Note: REST was conceived for working with information and media

Position: The Web is REST, so why not build services that way?
Resources

- Key concept in REST is the “resource” or “source of information”
- Each resource is referenceable via a global identifier (e.g. URI)
- Resource manipulation via standardized interfaces (e.g. HTTP)
- Resource exchange via documents representing the information
- Note: REST was conceived for working with information and media

*Position: The Web is REST, so why not build services that way?*
Resources

- Key concept in REST is the “resource” or “source of information”
- Each resource is referenceable via a global identifier (e.g. URI)
- Resource manipulation via standardized interfaces (e.g. HTTP)
- Resource exchange via documents representing the information

Note: REST was conceived for working with information and media

*Position: The Web is REST, so why not build services that way?*
Resources

- Key concept in REST is the “resource” or “source of information”
- Each resource is referenceable via a global identifier (e.g. URI)
- Resource manipulation via standardized interfaces (e.g. HTTP)
- Resource exchange via documents representing the information
- Note: REST was conceived for working with information and media

*Position: The Web is REST, so why not build services that way?*
Resources

- Key concept in REST is the “resource” or “source of information”
- Each resource is referenceable via a global identifier (e.g. URI)
- Resource manipulation via standardized interfaces (e.g. HTTP)
- Resource exchange via documents representing the information
- Note: REST was conceived for working with information and media

*Position: The Web is REST, so why not build services that way?*
Why use REST?

- Reliance on caching leads to better response time and reduced server loads
- Improves server scalability by reducing the need to maintain session state
- Less client-side code: just need a browser
- Less dependence on software layers on top of HTTP
- Equivalent functionality to other communication methods
- No separate resource discovery mechanism: use hyperlinks
- More maintainable than RPC due to layering:
  - Document types (e.g., HTML) can evolve without breaking backwards- or forwards-compatibility
  - Capability for resources to add support for new content types without changing support for older ones
- See [http://www.xfront.com/REST-Web-Services.html](http://www.xfront.com/REST-Web-Services.html) for a simple example of a REST design
Why use REST?

- Reliance on caching leads to better response time and reduced server loads
- Improves server scalability by reducing the need to maintain session state
- Less client-side code: just need a browser
- Less dependence on software layers on top of HTTP
- Equivalent functionality to other communication methods
- No separate resource discovery mechanism: use hyperlinks
- More maintainable than RPC due to layering:
  - Document types (e.g. HTML) can evolve without breaking backwards- or forwards-compatibility
  - Capability for resources to add support for new content types without changing support for older ones
- See http://www.xfront.com/REST-Web-Services.html for a simple example of a REST design
Why use REST?

- Reliance on caching leads to better response time and reduced server loads
- Improves server scalability by reducing the need to maintain session state
- Less client-side code: just need a browser
- Less dependence on software layers on top of HTTP
- Equivalent functionality to other communication methods
- No separate resource discovery mechanism: use hyperlinks
- More maintainable than RPC due to layering:
  - Document types (e.g. HTML) can evolve without breaking backwards- or forwards-compatibility
  - Capability for resources to add support for new content types without changing support for older ones
- See http://www.xfront.com/REST-Web-Services.html for a simple example of a REST design
Why use REST?

- Reliance on caching leads to better response time and reduced server loads
- Improves server scalability by reducing the need to maintain session state
- Less client-side code: just need a browser
- Less dependence on software layers on top of HTTP
- Equivalent functionality to other communication methods
- No separate resource discovery mechanism: use hyperlinks
- More maintainable than RPC due to layering:
  - Document types (e.g. HTML) can evolve without breaking backwards- or forwards-compatibility
  - Capability for resources to add support for new content types without changing support for older ones
- See http://www.xfront.com/REST-Web-Services.html for a simple example of a REST design
Why use REST?

- Reliance on caching leads to better response time and reduced server loads
- Improves server scalability by reducing the need to maintain session state
- Less client-side code: just need a browser
- Less dependence on software layers on top of HTTP
- Equivalent functionality to other communication methods
  - No separate resource discovery mechanism: use hyperlinks
  - More maintainable than RPC due to layering:
    - Document types (e.g. HTML) can evolve without breaking backwards- or forwards-compatibility
    - Capability for resources to add support for new content types without changing support for older ones
- See http://www.xfront.com/REST-Web-Services.html for a simple example of a REST design

Why use REST?

Reliance on caching leads to better response time and reduced server loads

Improves server scalability by reducing the need to maintain session state

Less client-side code: just need a browser

Less dependence on software layers on top of HTTP

Equivalent functionality to other communication methods

No separate resource discovery mechanism: use hyperlinks

More maintainable than RPC due to layering:
- Document types (e.g. HTML) can evolve without breaking backwards- or forwards-compatibility
- Capability for resources to add support for new content types without changing support for older ones

See http://www.xfront.com/REST-Web-Services.html for a simple example of a REST design
Why use REST?

- Reliance on caching leads to better response time and reduced server loads
- Improves server scalability by reducing the need to maintain session state
- Less client-side code: just need a browser
- Less dependence on software layers on top of HTTP
- Equivalent functionality to other communication methods
- No separate resource discovery mechanism: use hyperlinks
- More maintainable than RPC due to layering:
  - Document types (e.g. HTML) can evolve without breaking backwards- or forwards-compatibility
  - Capability for resources to add support for new content types without changing support for older ones
- See [http://www.xfront.com/REST-Web-Services.html](http://www.xfront.com/REST-Web-Services.html) for a simple example of a REST design
Why use REST?

- Reliance on caching leads to better response time and reduced server loads
- Improves server scalability by reducing the need to maintain session state
- Less client-side code: just need a browser
- Less dependence on software layers on top of HTTP
- Equivalent functionality to other communication methods
- No separate resource discovery mechanism: use hyperlinks
- More maintainable than RPC due to layering:
  - Document types (e.g. HTML) can evolve without breaking backwards- or forwards-compatibility
  - Capability for resources to add support for new content types without changing support for older ones

See [http://www.xfront.com/REST-Web-Services.html](http://www.xfront.com/REST-Web-Services.html) for a simple example of a REST design
Why use REST?

- Reliance on caching leads to better response time and reduced server loads
- Improves server scalability by reducing the need to maintain session state
- Less client-side code: just need a browser
- Less dependence on software layers on top of HTTP
- Equivalent functionality to other communication methods
- No separate resource discovery mechanism: use hyperlinks
- More maintainable than RPC due to layering:
  - Document types (e.g. HTML) can evolve without breaking backwards- or forwards-compatibility
  - Capability for resources to add support for new content types without changing support for older ones
- See http://www.xfront.com/REST-Web-Services.html for a simple example of a REST design
Workflows

- It is unlikely that \( \exists \) a web service that does either
  - Exactly what is wanted—the matching problem
  - All that is wanted—the composition problem

- Service workflows: structured sequences of workflows
- Workflow languages: programming for services
  - BPEL = Business Process Execution Language (OASIS standard)
  - Defines a high-level *orchestration* language
  - BPEL program is a XML document
  - BPEL is essentially procedural:
    - Sequencing
    - Iteration
    - Fork + join
    - Exceptions
  - BPEL interpreter: service that accepts a XML document and processes it, causing the specified services to be called and the specified data to be staged between them
Workflows

- It is unlikely that $\exists$ a web service that does either
  - Exactly what is wanted—the matching problem
  - All that is wanted—the composition problem

- Service workflows: structured sequences of workflows

- Workflow languages: programming for services
  - BPEL = Business Process Execution Language (OASIS standard)
  - Defines a high-level *orchestration* language
  - BPEL program is a XML document
  - BPEL is essentially procedural:
    - Sequencing
    - Iteration
    - Fork + join
    - Exceptions

- BPEL interpreter: service that accepts a XML document and processes it, causing the specified services to be called and the specified data to be staged between them
Workflows

- It is unlikely that ∃ a web service that does either
  - Exactly what is wanted—the matching problem
  - All that is wanted—the composition problem
- Service workflows: structured sequences of workflows
- Workflow languages: programming for services
  - BPEL = Business Process Execution Language (OASIS standard)
  - Defines a high-level orchestration language
  - BPEL program is a XML document
  - BPEL is essentially procedural:
    - Sequencing
    - Iteration
    - Fork + join
    - Exceptions
- BPEL interpreter: service that accepts a XML document and processes it, causing the specified services to be called and the specified data to be staged between them
Workflow Enactment

- Textual workflow authoring is not a human-friendly activity
- Widespread adoption of graphical programming approaches
- Many tools providing:
  - Simple deployment of programs as services
  - Service discovery (locally or from registries)
  - Drag-and-drop construction of plumbing diagrams
Workflow Enactment

- Textual workflow authoring is not a human-friendly activity
- Widespread adoption of graphical programming approaches
- Many tools providing:
  - Simple deployment of programs as services
  - Service discovery (locally or from registries)
  - Drag-and-drop construction of plumbing diagrams
Workflow Enactment

- Textual workflow authoring is not a human-friendly activity
- Widespread adoption of graphical programming approaches
- Many tools providing:
  - Simple deployment of programs as services
  - Service discovery (locally or from registries)
  - Drag-and-drop construction of plumbing diagrams
Example Taverna workflow
Content

1. Agile Development
2. Design Patterns
3. Web Architectures
4. RIAs
5. Web Services
6. Summary
Summary

- **Software development:**
  - There is no right way, but practice (others) and experience (yours) are the best guides
  - Aim to deliver “just enough” functionality
  - Don’t let features run your life

- **Design patterns:**
  - Build on best practice
  - Breaks solution into more easily testable components
  - Remember to test frequently and automatically

- **RIAs:**
  - Where is the compute power? Where is the bandwidth?
  - Terminal + mainframes, desktops + servers, thin clients + fat clients
  - Same principle, different terminology: 10-15 year cycle
Summary II

■ Web Services:

■ WSDL provides a means to describe the operational interface to a service, but says nothing about function
■ SOAP provides a familiar programming model, but encode/decode costs are high
■ UDDI provides a way to search for available services, but search semantics currently quite limited
■ REST (it is argued) is a better fit with the web metaphor and is seeing increasing uptake—avoids (?) need for the above
■ Workflow languages and editors are enabling visual “programming in the large”
“Never believe one source”

- W3C Rich Web Clients: http://www.w3.org/2006/rwc/
- W3C Web Application Formats Working Group: http://www.w3.org/2006/appformats/
- CURL @ MIT: http://www.cag.lcs.mit.edu/curl/
- Model-View-Controller: http://en.wikipedia.org/wiki/Model_view_controller
- [Freeman et al., 2004] provides a convenient more accessible text on design patterns.
- YAWL: http://www.yawl-system.com/
Resources II

- Slides for Ch.28 of Deitel & Deitel (see moodle)
- UNSPSC: Universal Standard Products and Services Classification
- NAICS: North American Industry Classification System
Resources III

- D-U-N-S Number: Data Universal Numbering System
- SOAP v1.2 (2007) http://www.w3.org/TR/soap12/
- RESTful services: http://en.wikipedia.org/wiki/Representational_State_Transfer
Ch. 28 of [Deitel and Deitel, 2008] discusses practical aspects of web services, but this is a very much an evolving area and the web is probably a better first resource. O’Reilly publish a useful book on RESTful web services [Richardson and Ruby, 2007]

*Internet & World Wide Web How to Program.*
Pearson.

*Architectural Styles and the Design of Network-based Software Architectures.*

*Head First Design Patterns.*
O’Reilly.
References II
