Cardiff's School of Computer Science & Informatics

CM1102 "Web Applications"

Modern Web Development

Introduction to Spring Semester

Dr Natasha Edwards

Learning objectives

- Welcome to Spring semester
- Examine principles of modern web development
  - frontend vs backend
  - web system development lifecycle
  - design patterns: MVC
  - web frameworks: Django

- Understand the need for software configuration management (SCM)
- Examine main principles of Version Control (VC)
- Introduce 

Spring semester: TOPICS

Backend web development:

- Design patterns (e.g. MVC)
- Version control, git
- Databases
- Web development frameworks
  - Python-based

Fundamentals and innovation:

- Networks
- Security, privacy, legal and ethical issues
- Search engines and Page Rank
- Commercial and economic context of the web and web applications (incl. e-commerce)
- Web and innovation (e.g. Semantic Web, AI)

Spring semester: LOGISTICS

CM1102 - Plan 2018/19

(correct as of 28/01/19)

WEEK No.	TOPIC	LECTURE	LAB	HELP CLASS (drop-in, optional)	ASSESSMENT
1	**Modern backend web development:** Introduction to Spring semester				
2	**Version control & Git**				
3	**Databases, MySQL**				
4	**Development and deployment of websites using framework-based web development**				**CW hand out**
5					
6					
7	**Networks**				
8	**Security**				**CW submission**
9	**Commercial and economic context of the web and web apps**				
10	- **Search engines** - **Innovation and the future**				
11	***Revision***				
12	***TBC:*** *Extra revision/ drop-in*				

***Note**: order of topics in Wks 7 - 10 are subject to change due to guest lecturers' availability*

Spring semester: ASSESSMENT

Coursework:

- Hand out: Week 4
- Submission: 9:30 am on Friday, 22nd ~~February~~ (Week 8)

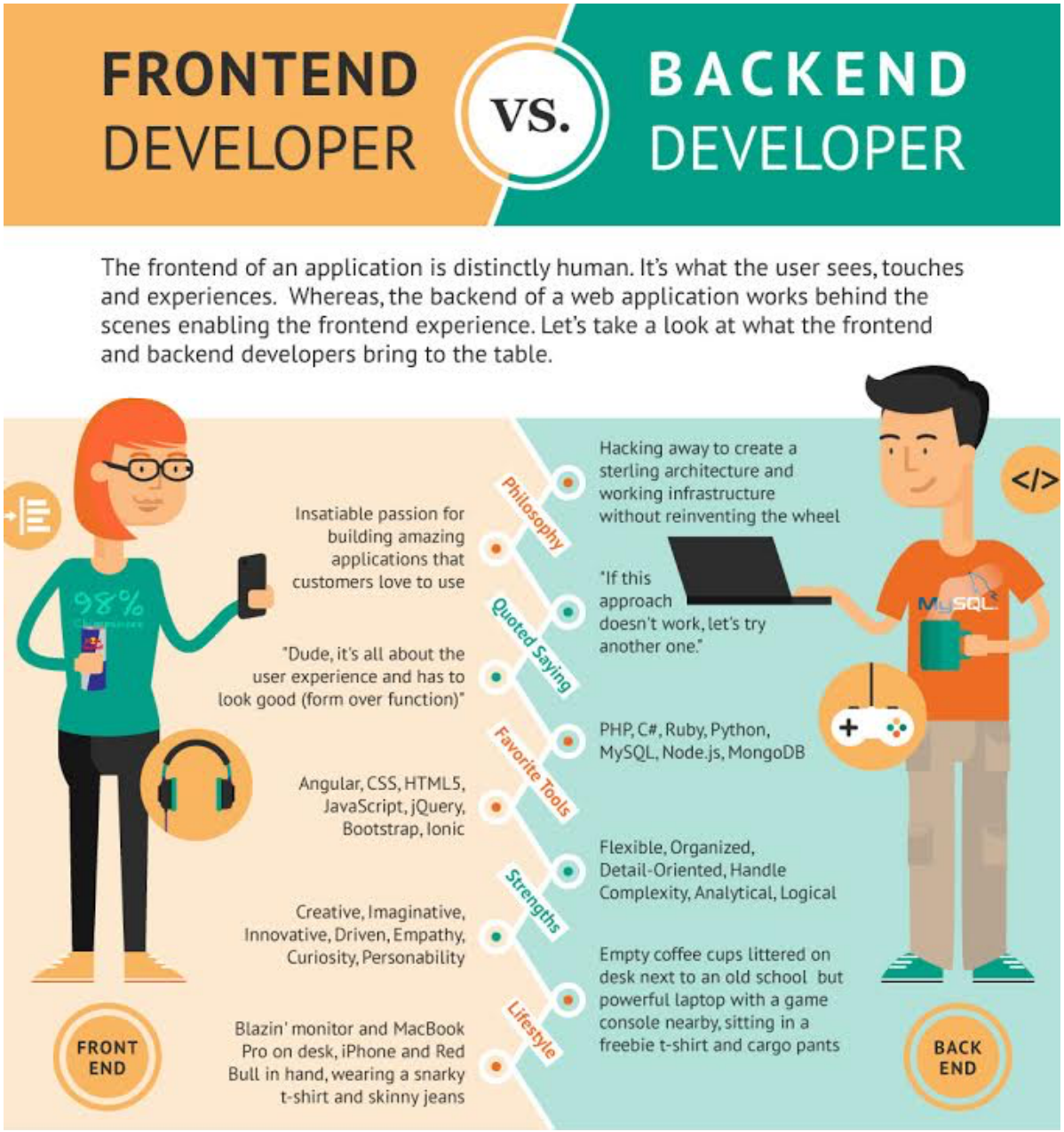
March !!

Exam:

- 2 hr, 40% of the module

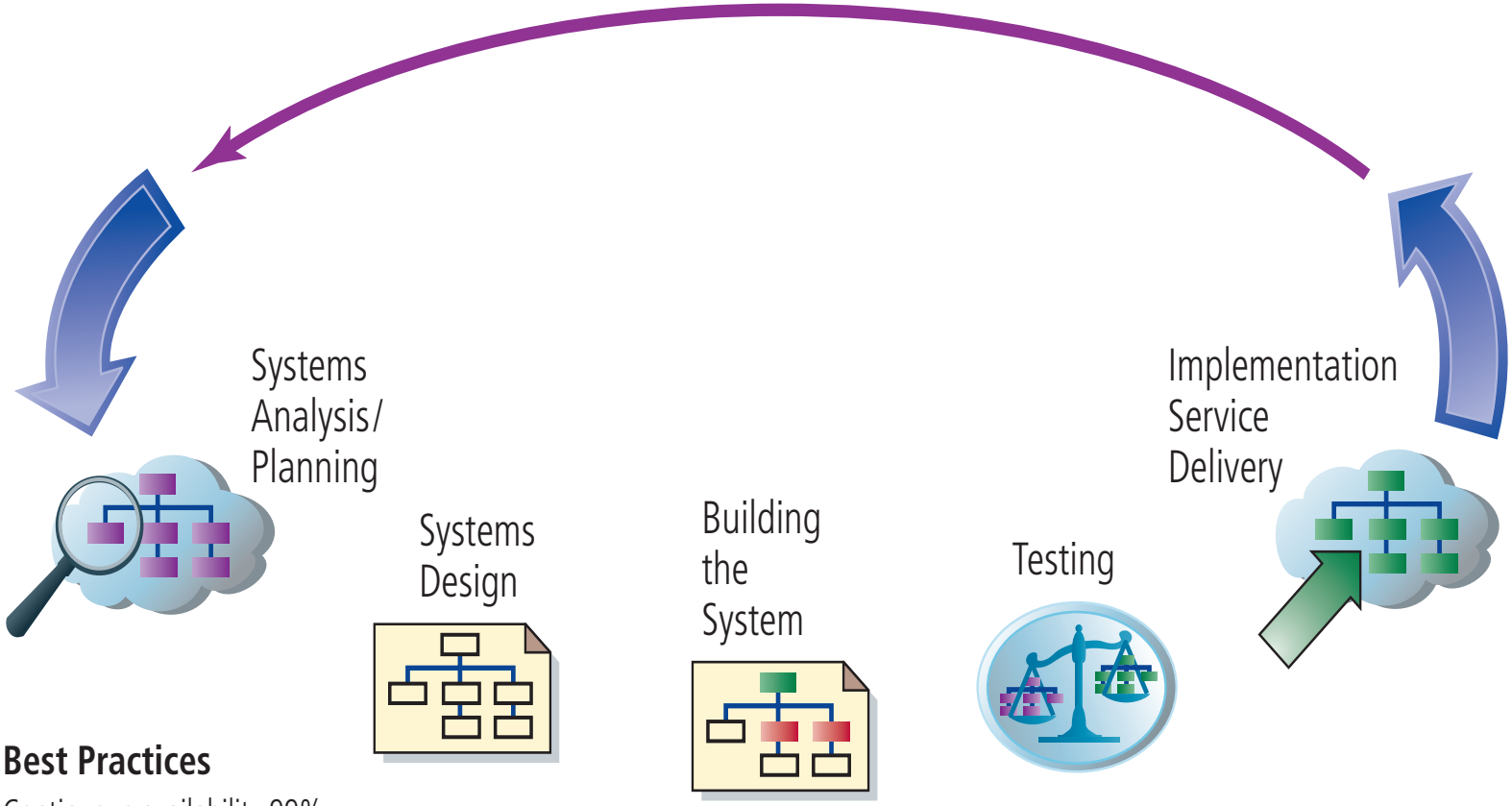
Principles of modern web development

Frontend vs. Backend development



(Source: www.linkedin.com)

Web systems development lifecycle



Best Practices

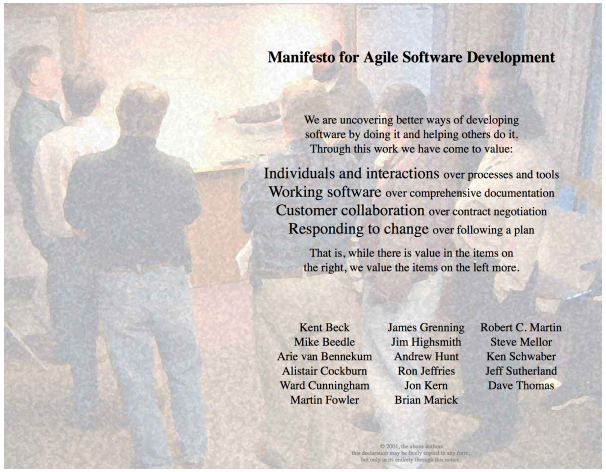
Continuous availability 99%+
Design for scalability
Build in management for end-to-end delivery
Plan for growth
Design pages for high-speed performance
Understand and optimize workload on system

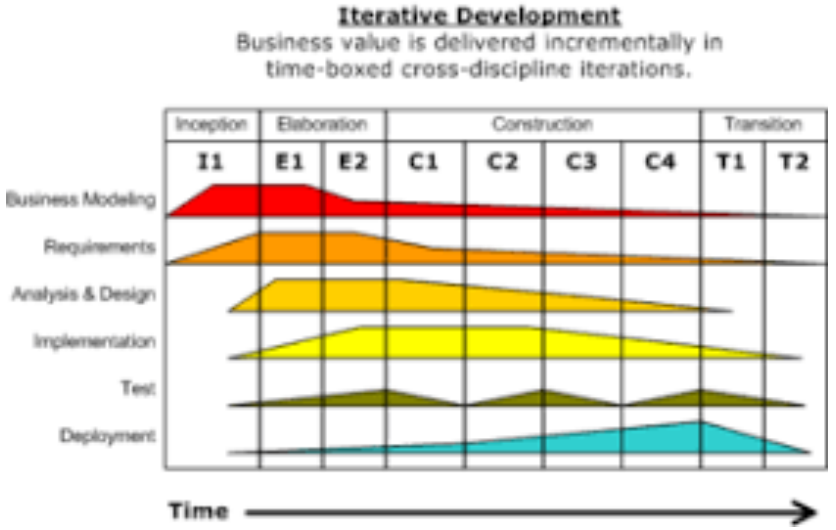
Also, maintenance!

All stages involves changes and it is important to manage these effectively and efficiently

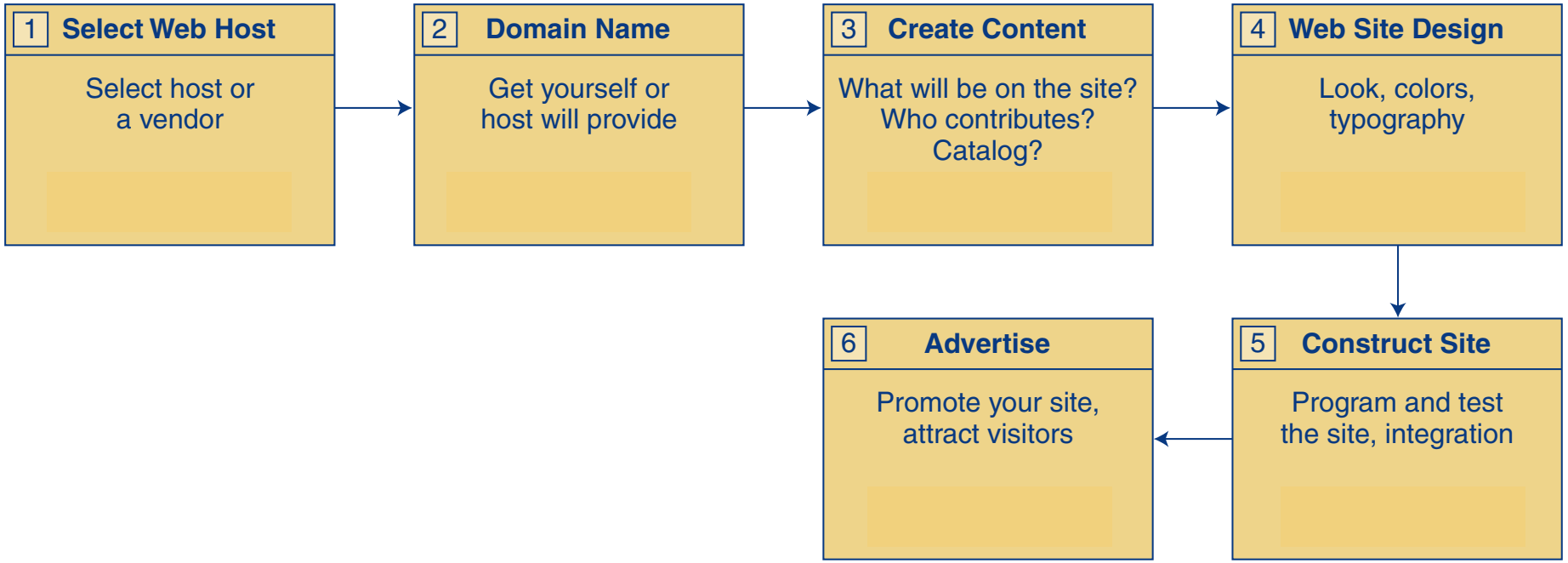
Other approaches

- Agile (XP, Scrum, Kanban…)
  - https://www.agilealliance.org
  - Agile Manifesto: https://agilemanifesto.org
- **R**ational **U**nified **P**rocess (RUP)

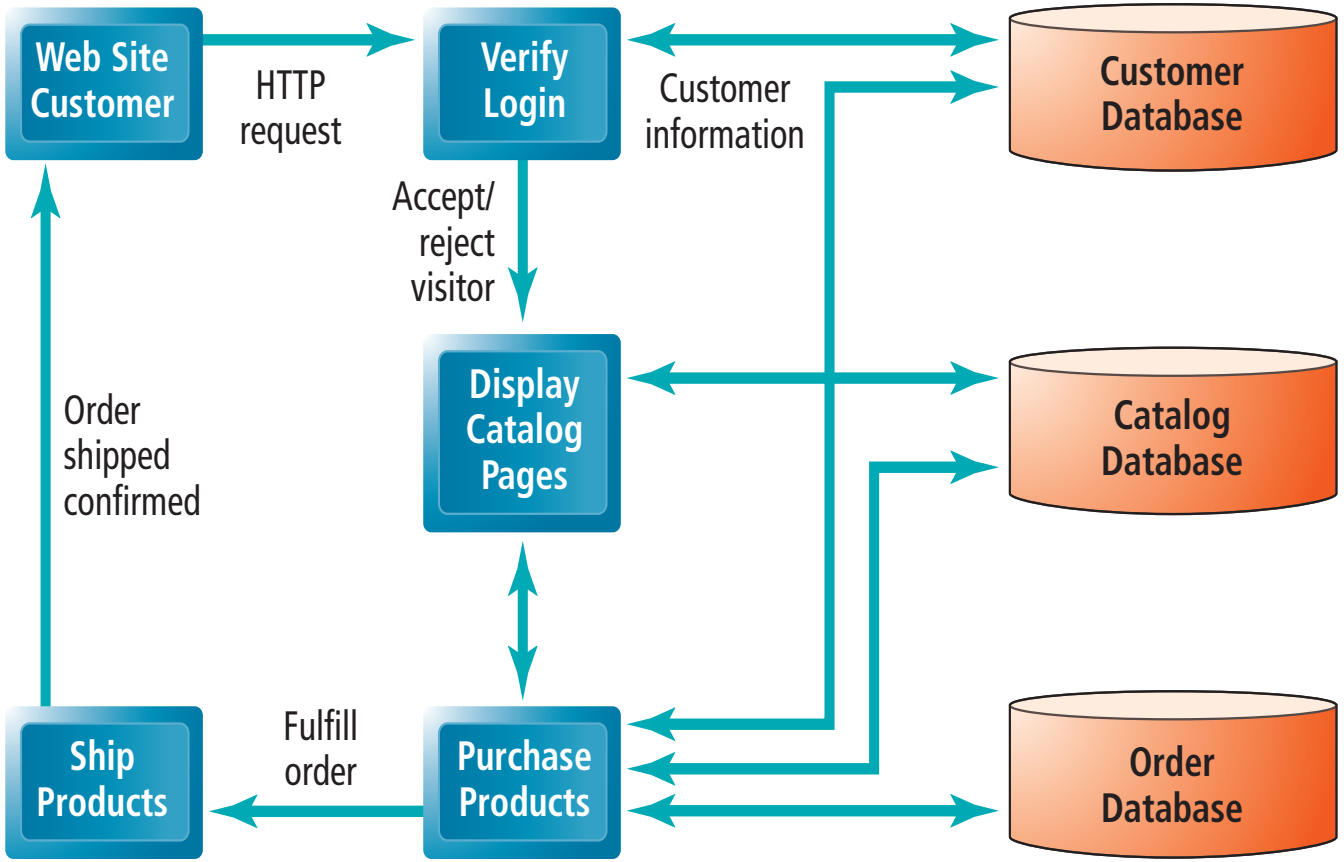




Process of building a website



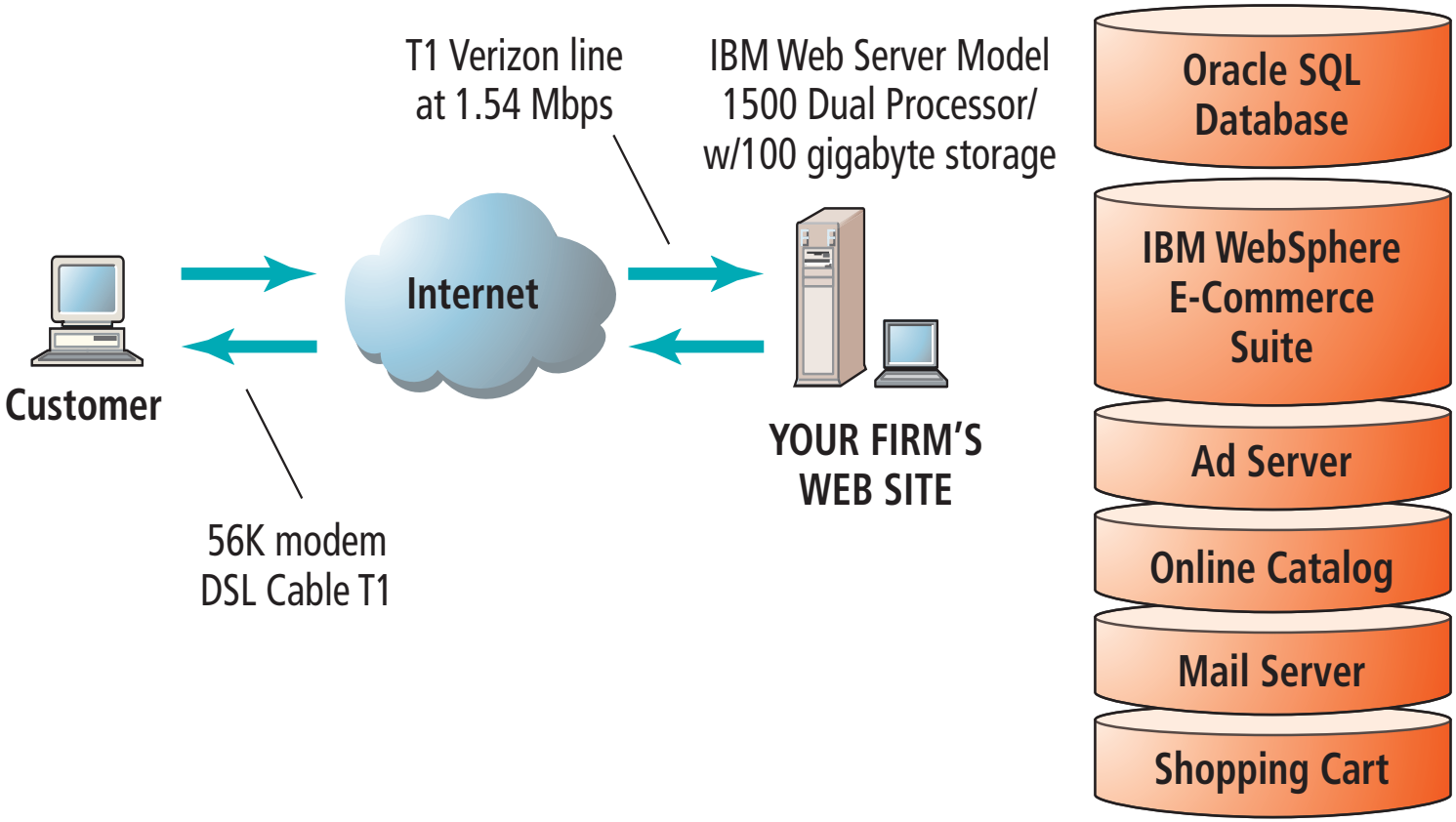
Logical design of a simple site



(a) Simple Data Flow Diagram

This data flow diagram describes the flow of information requests and responses for a simple Web site.

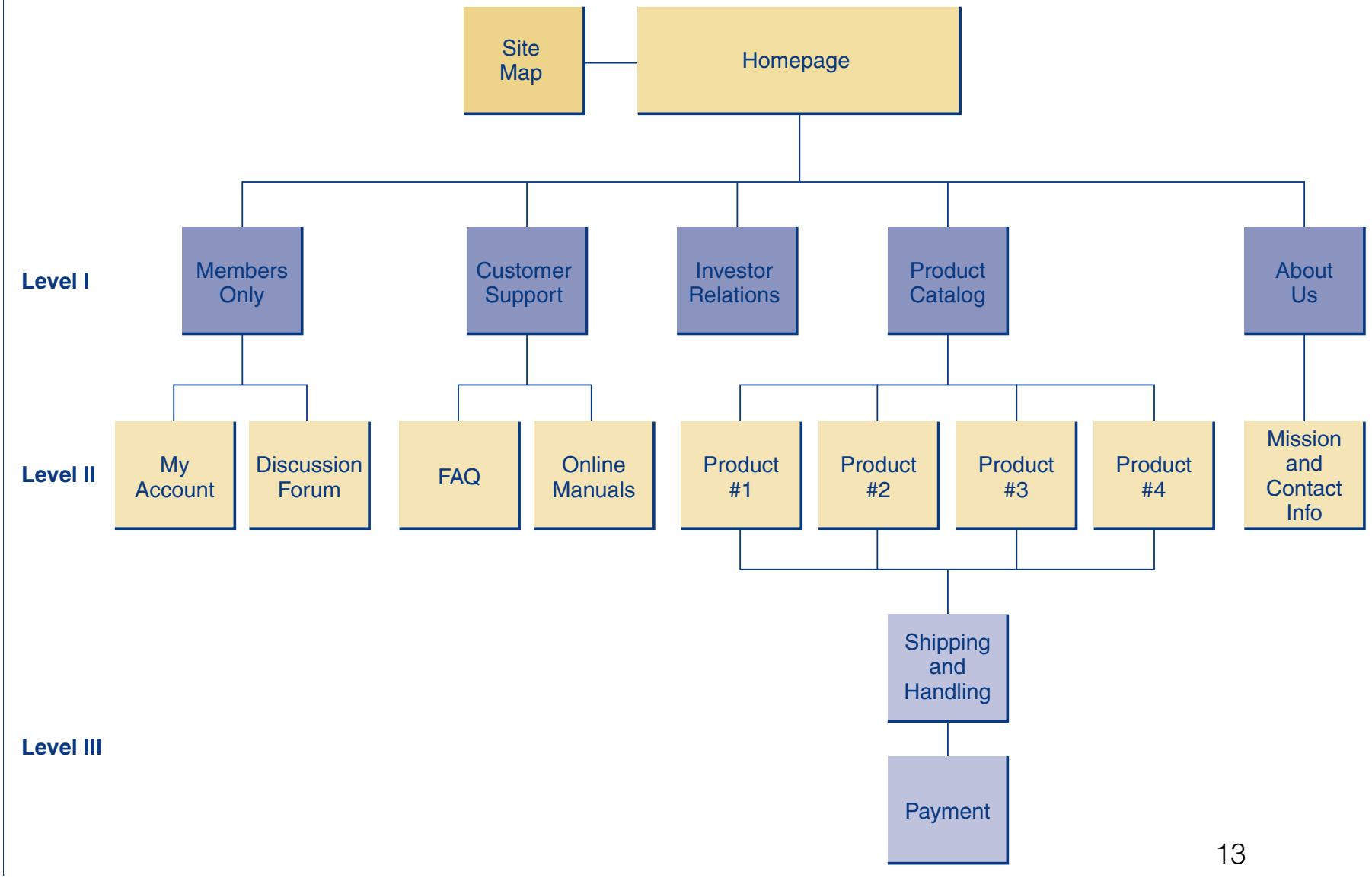
Physical design of a simple site



(b) Simple Physical Design

A physical design describes the hardware and software needed to realize the logical design.

Information architecture of a simple site



Design patterns

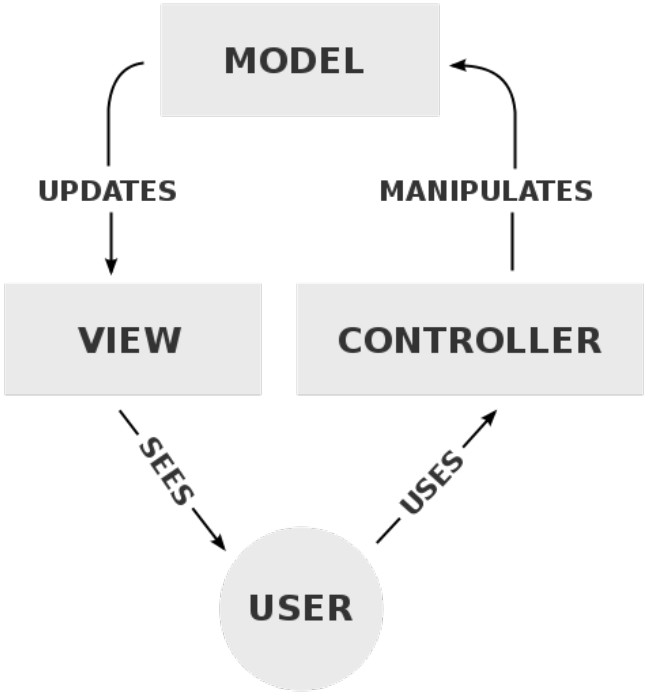
- Reusable solutions to a common problem and within a context
  - *Creational*, *structural*, *behavioural*, *concurrency*, *architectural*, …

Pattern: Model-View-Controller

Model-**V**iew-**C**ontroller is an extremely common **architectural pattern** in software engineering.

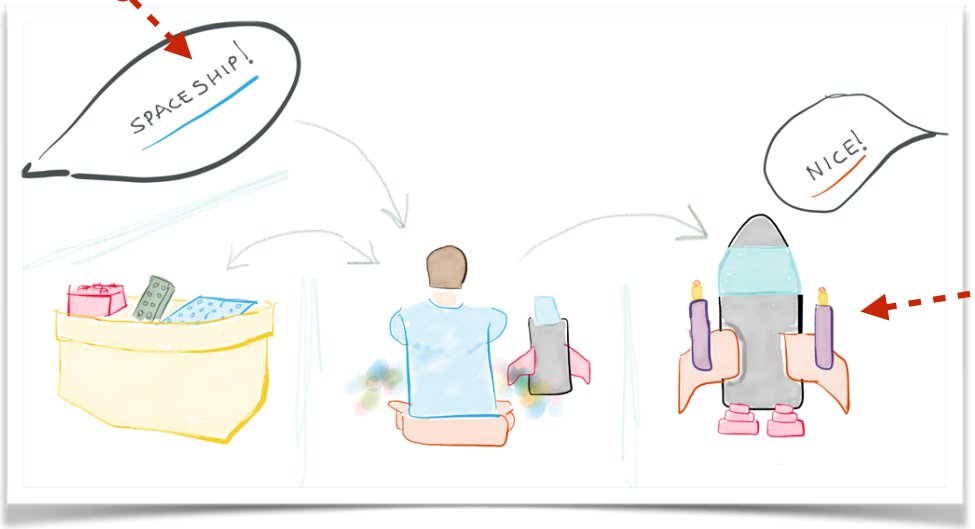
It describes a way to logically structure an (**interactive**) application and how to separate the responsibilities and interactions for its parts.

- **Model**: maintains state and notifies views and controllers when change of state occurs.
- **View**(s): request information from the model and present it to the user.
- **Controller**(s): updates the model state in response to the user's actions, and possibly updates views.



Lego & MVC

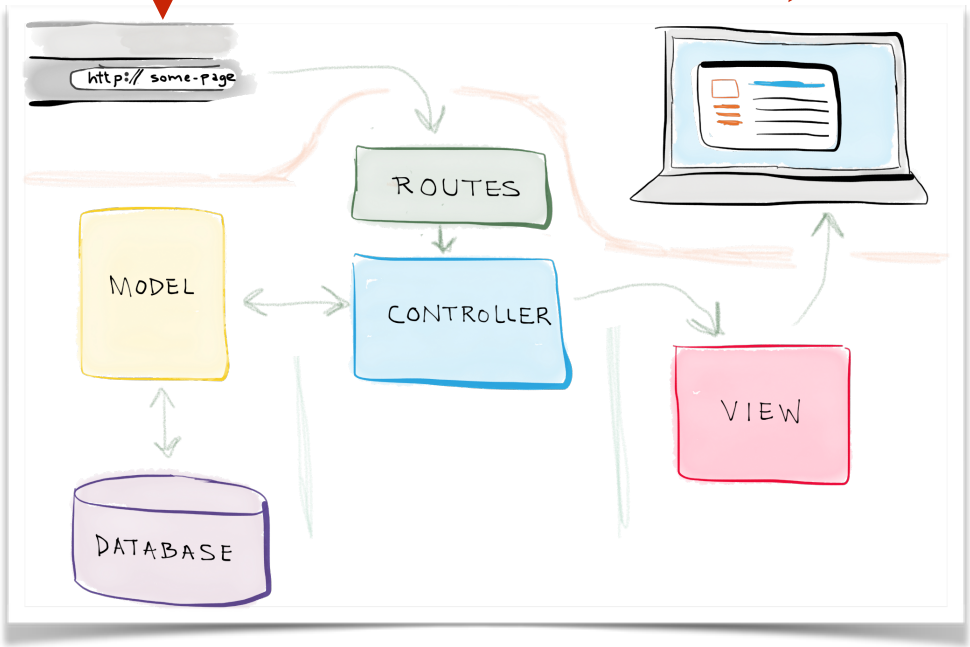
request



response

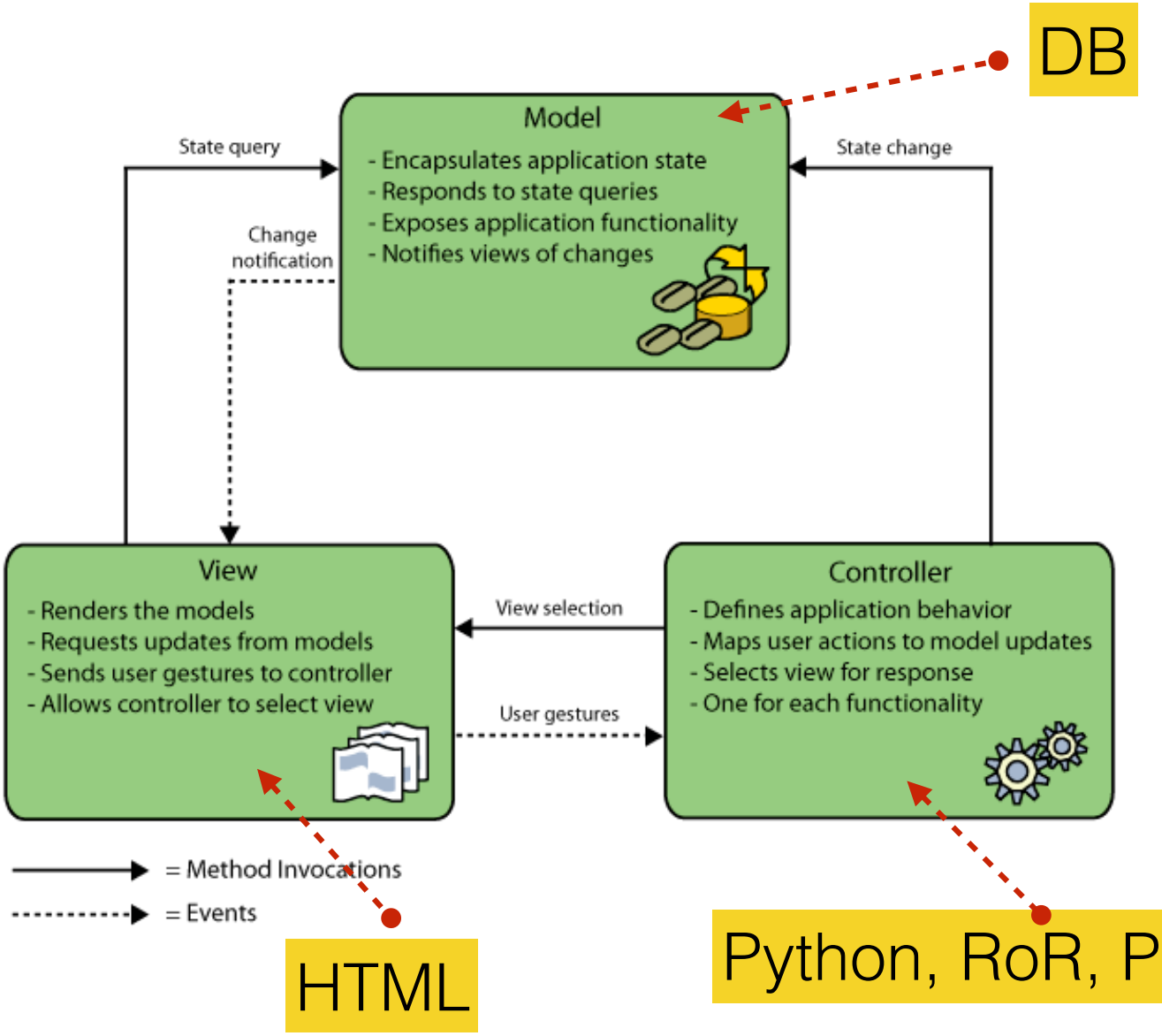
request

response



Source: https://realpython.com/the-model-view-controller-mvc-paradigm-summarized-with-legos/

Architectural Patterns: MVC



Why?

- easier to develop *(modular; team-working)*
- easier to maintain, with cleaner code
- scalable

Pattern: Model-View-Controller

Examples of situations that can be modelled by **MVC**:

- Computer games.
- Text editors, and countless other GUIs.
- Various interactive web sites.
- Hardware interfaces
  (*e.g.* nuclear power plant control panel).

Web Frameworks

In software engineering, some problems, scenarios, situations, patterns are **frequently recurring**.

- Express solutions to common problems as libraries.
- **Reuse** code in the future.
- Do not reinvent the wheel:
  - Not only to **save time**, but to **avoid poor in-house solutions**.
- **D**o not **R**epeat **Y**ourself (DRY principle).

Web Frameworks

Examples of common problems in **web programming**:

- **Storing state** of a website in a systematic manner
  - Abstracting from the database engine.
  - Mapping between relational databases and classes/ objects.
  - Maintaining some business logic (e.g. decrement stock counter when an item is sold).
- **Presenting** data to users according to some business logic.
- **Reacting** to user's requests/actions, for example to modify the **state** or **present data** in a different way.

(you most likely have encountered some of these already)

Web Frameworks

Some **more specific examples** of patterns in web programming:

- Authentication of users
- Handling sessions
- Form validation
- Taking payments
- Searching.
- URL routing (mapping URLs to actions).
- Skinnable HTML interfaces, e.g. via templates.
- Caching to reduce server load and increase responsiveness.

. . . and countless more!

Web Frameworks

Web Application Frameworks (or simply **"web frameworks"**) aim to address the above common problems, patterns, scenarios.

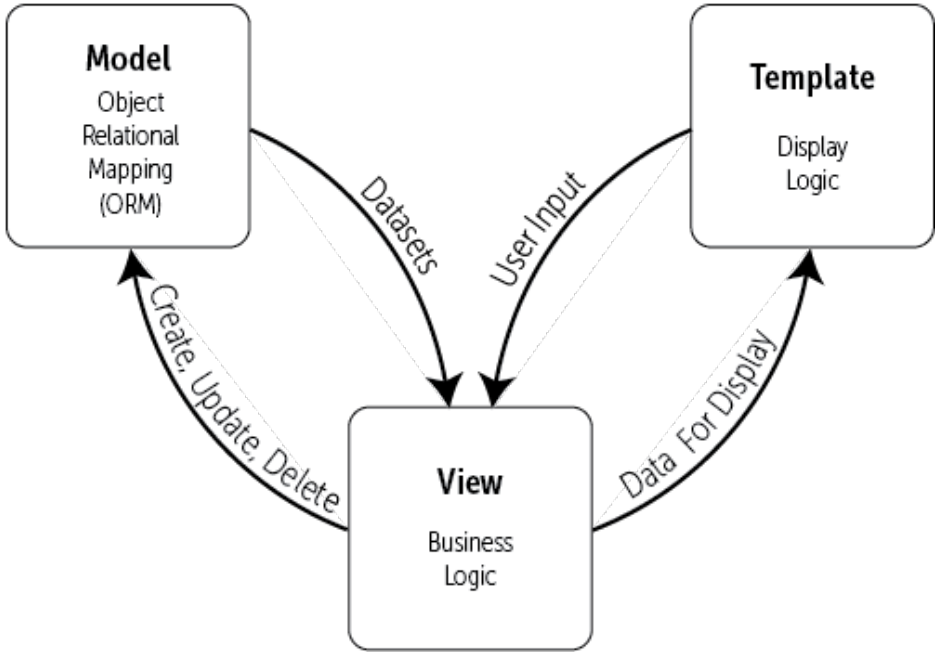
- Huge list of existing web frameworks:

  http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks
- Available for many languages. Some popular ones:
  - **Ruby on Rails** (Ruby)
  - **Flask, Django** (Python)
  - **weblocks** (Lisp)
  - **Yii** (PHP)
- Various level of abstractness. Some solve specific problems, some are very general.
- We will introduce **Django**, but will use **Flask** *(labs in Weeks 4 - 6)*

Django framework & MVC

MVC = MTV in Django

- **M** is for **"Model"**:
  - the data access layer, which contains anything and everything about the data: how to access it, how to validate it, which behaviour(s) it has, and the relationships between the data (i.e. description of your data structure and db schema)
- **T** is for **"Template"**
  - the presentation layer, which contains presentation-related decisions: how content should be displayed on a page or other type of document.
- **V is for "View":**
  - the business logic layer, which contains the logic that accesses the model and defers to the appropriate template(s). Consider this as the bridge between models and templates.



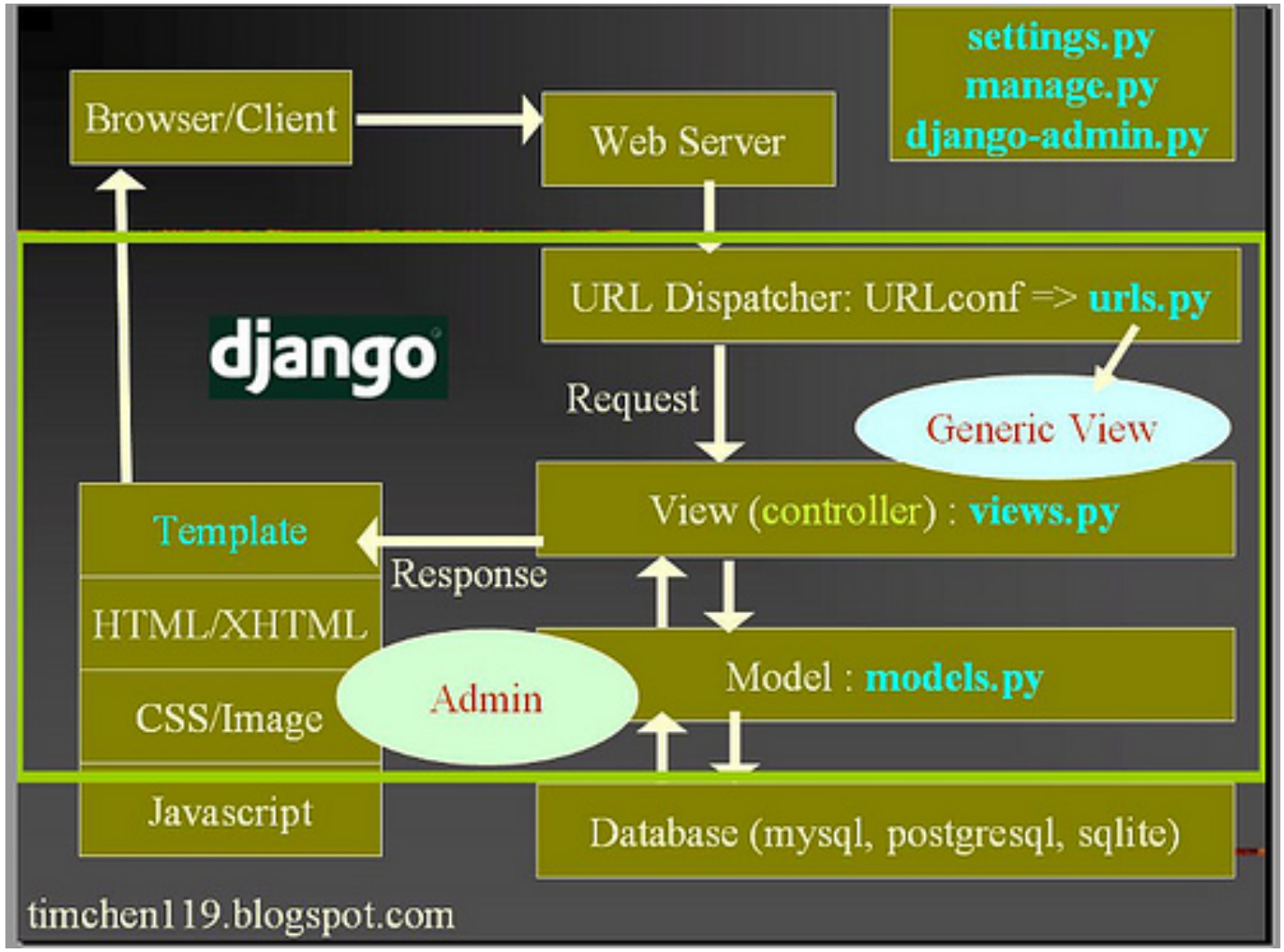
Source:
https://djangobook.com/tutorials/django-overview/

Django modules

- Admin interface (CRUD interface)
- Forms handling
- Internationalisation, localization
- Syndication framework (RSS and Atom)
- Authentication
- Sessions
- Comments
- Caching
- Custom Middleware

Full list: https://docs.djangoproject.com/en/2.2/py-modindex/

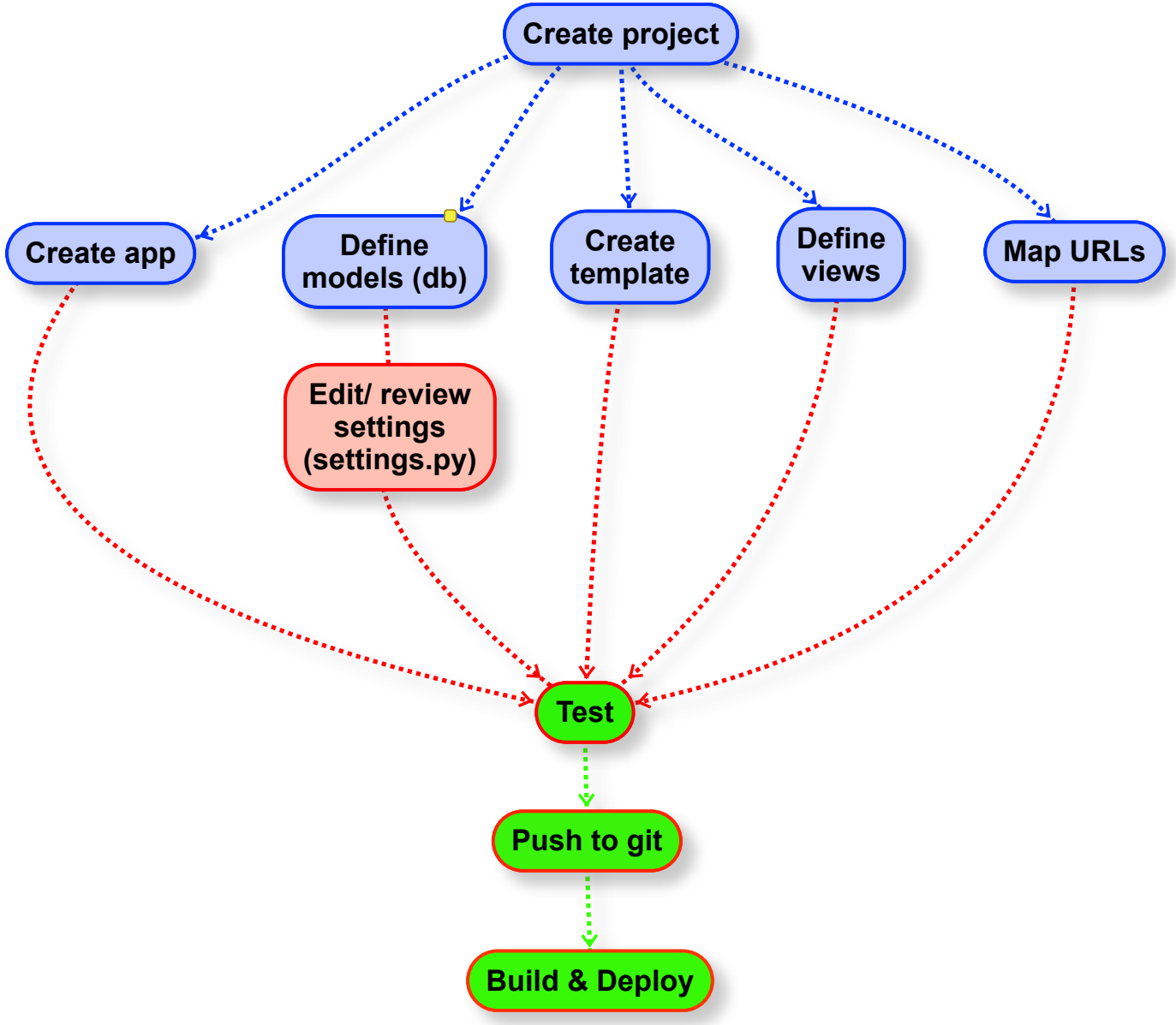
Django Architecture



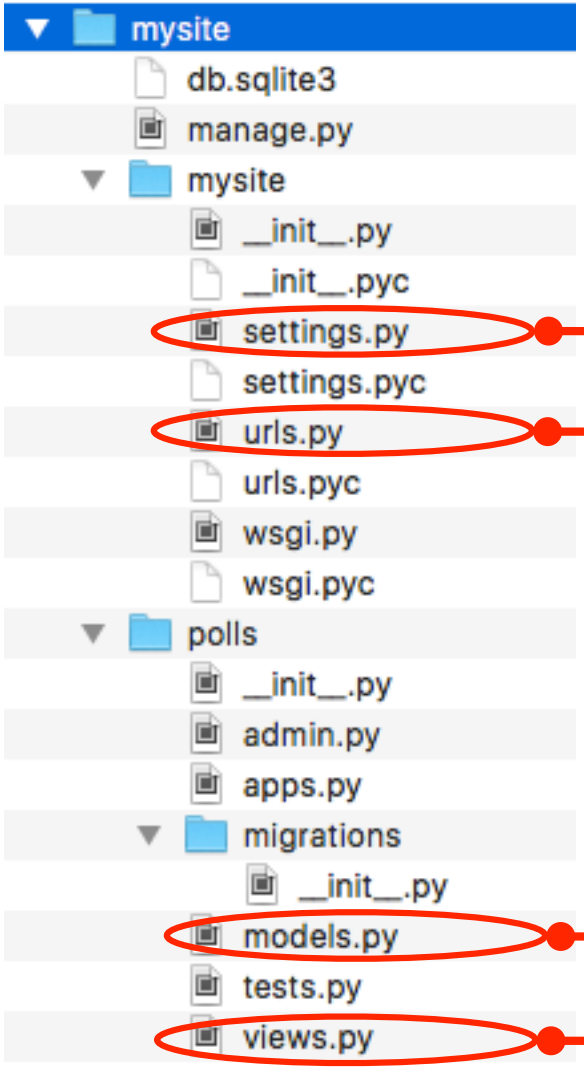
Projects vs Apps

- Project is the entire ‘program’ (e.g. the whole website); can be viewed as a collection of apps and configurations to create a ‘website experience’.
- App is a module, which provides some specific functionality.
  - Apps can be reused in other projects.
    - *e.g. a blog, user authentication, social media integration, user polls/ surveys, …*
  - NB: Can’t run an app without a project

Django project: basic workflow



An example Django project



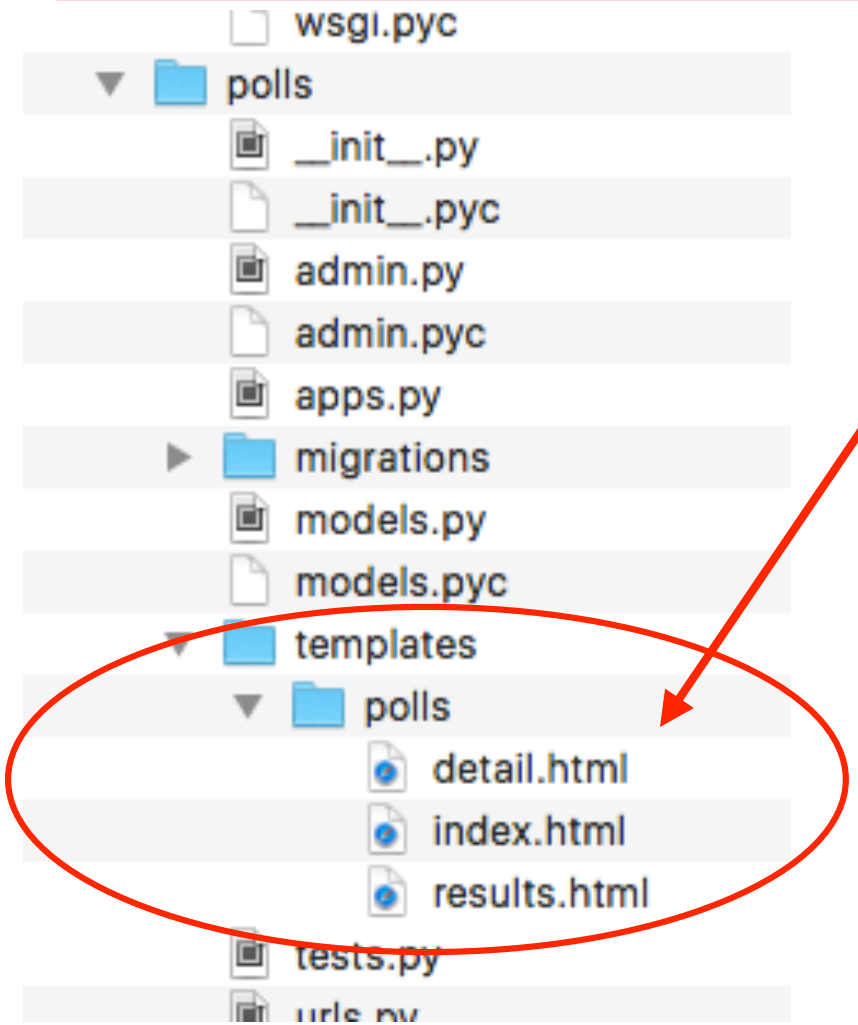
central configuration for a **Django** project

mapping between URL path expressions to Python functions (your views)

DB

takes a Web request and returns a Web response

An example Django project




```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

e.g. variable **`{{ section.title}}`** will be replaced with the **`title`** attribute of the **`section`** object

An example Django project

polls/models.py

models.py

```
1  from django.db import models
2  import datetime
3  from django.utils import timezone
4
5
6
7  class Question(models.Model):
8      question_text = models.CharField(max_length=200)
9      pub_date = models.DateTimeField('date published')
0
1      def __str__(self):
2          return self.question_text
3  # ...
4      def was_published_recently(self):
5          now = timezone.now()
6          return now - datetime.timedelta(days=1) <= self.pub_date <= now
7      was_published_recently.admin_order_field = 'pub_date'
8      was_published_recently.boolean = True
9      was_published_recently.short_description = 'Published recently?'
0
1
2  class Choice(models.Model):
3      question = models.ForeignKey(Question, on_delete=models.CASCADE)
4      choice_text = models.CharField(max_length=200)
5      votes = models.IntegerField(default=0)
6
7      def __str__(self):
8          return self.choice_text
```

An example Django project

polls/urls.py

need to create **urls.py** file in **polls/** dir

```
from django.conf.urls import url

from . import views

urlpatterns = [
  url(r'^$', views.index, name='index'),
]
```

polls/views.py

views.py

```
1  from django.http import HttpResponse
2
3  def index(request):
4      return HttpResponse("Hello, world. You're at the polls index.")
5
```

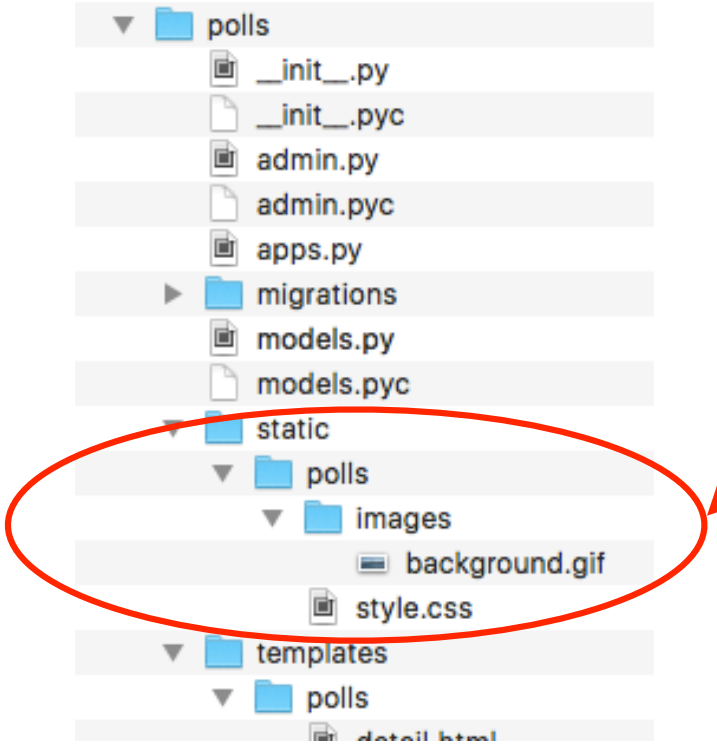
Adding 'Static' (images, CSS, JS, etc.)

```
polls/static/
```

```
polls/static/polls
```

```
polls/static/polls/images
```

create folders



Software Configuration Management
&
Version Control

What is Configuration Management?

Configuration Management is the process of identifying and defining the items in the system, controlling the change of these items throughout their lifecycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items.

IEEE Guide to Software Configuration Management 1987.

(IEEE/ANSI Standard 1042-1987)

Software Configuration Management (SCM)

- is a quality assurance activity that manages change throughout the software process
- controls, manages and documents change in software systems
- coordinates the creation and release of products to customers

Problems of scale

Development of a large software system:

- Involves a large amount of components
  - Requirements, designs, code modules, test cases, documentation etc.
  - Complex dependencies between components
  - Software system undergoes many changes
- May involve large teams of developers
  - Possibly dispersed amongst a number of different sites.
- Some of the components may be shared
  - between developers
  - between systems (reuse)

Need for Configuration Management

Number of areas of possible **conflict:**

- **Simultaneous Update**
  - Two (or more) programmers each take a copy of the same component and make a change
  - Last copy submitted overwrites changes made previously
- **Changes to Shared Code**
  - Need to check changes do not conflict with others who use the item
- **Keeping Variants Consistent**
  - To support different platforms or operating systems

Version Control (VC)

- a.k.a. Revision Control, a.k.a. Source Control
- is a part of SCM
- Management of change:
  - Track changes, backup and synchronise files
- VCS = Version Control System

Why Version Control?

- Imagine you are working on a project (coursework) and making lots of revisions.
- Or imagine you are working on a project that involves a group of people. Now imagine that several people working on the same document, how do you know who produced/ changed what and what was the latest change?
- VC = Time Machine:
  - Allows you to go back in time to the previous working version
- Can be a project requirement

Who is it for?

Types of Version Control

Individuals	→	Individual VC
Groups (collaboration)	→	Remote, Distributed VC

Version Control Systems

- Version Management is a technique that facilitates Change Control
- It captures and controls changes to each of the components that make up the system
  - A new version may be created:
    - As an initial version when a new component is first created
    - As a fix for a problem
    - To add extra functionality
    - To meet a change in requirements
    - As a variant e.g. for multiple platform support

Version Control Systems /cont'd

- A repository is used to store components
- Developers check out a copy of the component into their workspace
  - Provide exclusive locking of the component
  - Or allow changes to be merged
- Once the changes are made the component can be checked in to the repository.
  - Metadata may also be stored:
    - e.g. the author, the date and a brief description
    - This builds up a version history for the component

What VCS can do

- A versioning system may provide facilities for:
  - storing and retrieving past versions of components and products (back up and restore)
  - components (files) synchronisation
  - short-term and long-term undos
  - creation of new versions from old
  - sandbox changes (e.g. by branching and then merging/ rebasing)
  - checking the differences between two versions of the same component

What VCS can do /cont'd

- A versioning system may provide facilities for:
  - tracking changes and ownership
  - automatic management of version identifiers
  - documentation of change histories
  - automatic retrieval of all code in a particular release
  - providing notification of changes

Advantages and Disadvantages

+ Sandbox for everyone
+ Works offline
+ Fast
+ [Fairly] easy
+ Less Management
- Still need backups
- What is the "latest" version?
- No "proper" revision numbers

VC Lingo

- **Repository (repo)**: The DB to store the files (objects)
- **Server**: to store the repo
- **Client**: to connect to the repo
- **Working set/ working copy**: your local directory of files, where you make changes
- **Trunk/ Main**: the primary location for code in the repo. Think of code as a family tree — the trunk is the main line
- **Global unique identifiers** (uses SHA-1).
  - *NB: Files with identical content will have the same SHA-1*

Using Git for version control

Lab next week!